TCG Storage Architecture Core Specification

Specification Version 1.0
Revision 0.9 – draft –
May 24, 2007
Draft

Work In Progress
This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

TCG

Copyright © TCG 2007

Copyright © 2003-2007 Trusted Computing Group, Incorporated.

Disclaimer

THIS SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Without limitation, TCG disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

No license, express or implied, by estoppel or otherwise, to any TCG or TCG member intellectual property rights is granted herein.

Except that a license is hereby granted by TCG to copy and reproduce this specification for internal use only.

Contact the Trusted Computing Group at www.trustedcomputinggroup.org for information on specification licensing through membership agreements.

Any marks and brands contained herein are the property of their respective owners.

Revision History

a	Added Sections	R. Thibadeau
b	8/25/04 Extensive Edits	R. Thibadeau
c	8/31/2004 Lot of changes, particularly types	J. Nestor
d	9/7/04 Added more explanation of SPs	R. Thibadeau
e	9/16/04 Changed Intro, included example	R Thibadeau
f-g	8/5/05 Numerous Edits, Filled out ACL Section	R. Thibadeau, D. Philips, J. Cox
h	8/31/05 More ACL Section work	R. Thibadeau, D. Philips, J. Cox
j	9/8/05 Slight ACL update, Session update	R. Thibadeau, D. Philips, J. Cox
J	9/9/05 Secure Messaging	R. Thibadeau, D. Philips, J. Cox
k-l	9/12/05 Misc. cleanup	R. Thibadeau, D. Philips, J. Cox
m	9/14/05 Secure Messaging diagrams and cleanup.	R. Thibadeau, D. Philips, J. Cox
n	9/22/05 Additional Secure Messaging diagrams.	R. Thibadeau, D. Philips, J. Cox
o-q	10/11/05 Addt'l. core spec details & org. changes.	R. Thibadeau, D. Philips, J. Cox
p	10-18-05 More small edits for clarity	R. Thibadeau
r-s	11/04/05 Changes from October Orlando F2F	R. Thibadeau, D. Philips, J. Cox
t	11/xx/05 Changes from post F2F conference calls	R. Thibadeau, D. Philips, J. Cox
u-v	11/23/05 Changes from Working Group Core Comments, and cleanup of text, small edits, etc.	R. Thibadeau, D. Philips, J. Cox
w	12/12/05 Removed SP->SP method calls, clarified logging during a read session, etc., some updates from the SF F2F in Dec '05	R. Thibadeau, D. Philips, J. Cox
0.2a-b	12/22/05 Moved up to 0.2 as we're adding fewer and fewer new issues.	R. Thibadeau, D. Philips, J. Cox
0.2c-d	1/5/06 Added Life Cycle (Section 10)	R. Thibadeau
0.2e-i	Numerous clarifying changes as per TCG comments	R. Thibadeau, D. Philips, J. Cox
0.3	Numerous clarifying changes as per TCG comments, updated EC tables, added curve choices	R. Thibadeau, D. Philips, J. Cox, D. Brown
0.4	4/7/06 Major Format & Section Changes. Added Registry & Locking SP sections	D. Ybarra
0.5	Numerous small edits filling in gaps left in reformat	R. Thibadeau
0.6	Fixed numerous doc bugs, added section on ECMQV	J. Cox, D. Brown
0.7	Many changes, doc bug fixes, clarifications, and updates based on WG comments and Triage	J. Cox
0.9	Significant modifications based on SWG document review, multiple proposal submissions, etc. in preparation for SWG vote	J. Cox and the rest of the SWG

TABLE OF CONTENTS

Figures

Tables

1 Introduction

1.1 Scope and Audience

The Storage Workgroup specifications are intended to provide a comprehensive command architecture for putting selected features of storage devices under policy-driven access control. The capabilities of the storage device can be configured to conform to the policies of the trusted platform. In accord with the Storage Workgroup Use Cases and Peripherals Workgroup Use Cases documents, the controlled features include access to secure storage areas and the lifecycle state of the storage device as a trusted peripheral (TPer). This document may also serve as a specification for TPers where that is deemed appropriate.

The intended audience for this document is storage device and peripheral device manufacturers and developers that may wish to tie storage devices and peripherals into trusted platforms.

The following table lists the primary topics contained in this specification:

Table 01 Core Architecture Topics

Component	Function
Data definitions	Basic data types
Templates	Types of Security Providers (SPs) and their roles
Table definitions	Table's purpose (& Security Associations)
Methods	Commands purpose & data structures
Access Control	Authority model for Access Control
Sessions	Command streams
Secure Messaging	Authenticated Confidential Command Streams
SP Issuance and Personalization	Creating and Deleting SPs for custom uses
Reference Manual	Formal definitions for each SP, Table, and Method
Life Cycle	Default Table States, State Transitions, and Access Controls

1.2 Key Words

Key words are used to signify the requirements in the specification. The key words "shall," "should," "may," and "optional" are used in this document. These words are a subset of the RFC-2119 key words used by TCG, and have been chosen since they map to key words used in T10/T13 specifications. These key words are to be interpreted as described in [RFC-2119].

1.3 References

TCG Storage Workgroup Use Cases

T10 SCSI SECURITY PROTOCOL IN/OUT Commands, SCSI Primary Commands draft SPC04r05 or later

T13 ATA TRUSTED SEND/RECEIVE Commands, ATA8 Commands draft T13/1699-D Rev 3c or later

TCG Storage Certificates Specification

TCG Common Criteria Security Target – **Note that the quality of random numbers and cryptographic computations is the purview of the CC Security Target, not this specification.**

Serial ATA 2.6 (SATA-2). 15 February 2007 – **Note that for information on the current status of Serial ATA documents, see the Serial ATA International Organization at http://www.sata-io.org.**

ISO/IEC 14776-871. *AT Attachment – 8 ATA/ATAPI Command Set (ATA8-ACS)*(ANSI INCITS T13/1699D)

ISO/IEC 14776-151, *Serial Attached SCSI 1.1 (SAS-1.1)*(ANSI INCITS 417-2006)

ISO/IEC 14776-312, *SCSI Primary Commands - 3 (SPC-3)*(ANSI INCITS 408-2005)

Internet Engineering Task Force, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile (RFC 3280)

Internet Engineering Task Force, Character Mnemonics & Character Sets (RFC 1345)

National Institute of Standards and Technology (NIST), *Security Requirements for Cryptographic Modules*, FIPS Publication 140-2, May 25 2001

National Institute of Standards and Technology (NIST), *Secure Hash Standard*, FIPS Publication 180-2, August 1 2002

National Institute of Standards and Technology (NIST), *Digital Signature Standard (DSS)*, FIPS Publication 186-2, January 27 2000

FIPS Publication 186-3 (Draft revision of FIPS 186-2)

National Institute of Standards and Technology (NIST), *Advanced Encryption Standard (AES)*, FIPS Publication 197, November 26 2001

National Institute of Standards and Technology (NIST), *The Keyed-Hash Message Authentication Code (HMAC)*, FIPS Publication 198, March 6 2002

National Institute of Standards and Technology (NIST), *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*, NIST Special Publication 800-38A, December 2001

National Institute of Standards and Technology (NIST), *Recommendation for Block Cipher Modes of Operation –The CMAC Mode for Authentication,* NIST Special Publication 800-38B, May 2005

National Institute of Standards and Technology (NIST), *Recommendation for Block Cipher Modes of Operation – The CCM Mode for Authentication and Confidentiality*, NIST Special Publication 800-38C, May 2004

National Institute of Standards and Technology (NIST), *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) Mode for Confidentiality and Authentication*, NIST Draft Special Publication 800-38D, April 2006 (see http://csrc.nist.gov/publications/drafts/Draft-NIST_SP800-38D_Public_Comment.pdf)

National Institute of Standards and Technology (NIST), *Recommendation for Pair-Wise Key Establishment Using Discrete Logarithm Cryptography*, NIST Special Publication 800-56A, March 2006

RSA Laboratories, *PKCS #1: RSA Cryptography Standard (v 1.5)*, November 1 1993

RSA Laboratories, *PKCS #1: RSA Cryptography Standard (v 2.1)*, June 14 2002

1.4 Terminology

1.4.1 Global Terminology

Table 02 Global Terminology

Term	Definition

Term	Definition
ACE	Access Control Element. Defined as Rows in an SP's ACE Table. This is a Boolean expression of Authorities and the associated Row and Column Restrictions on the Method/Table, Method/Object, or Method/SP combination to which the ACE is attached.
ACL	List of ACEs associated with a particular Method/SP, Method/Table, or Method/Object combination.
Admin	The predefined 'superuser' on any SP. The Transport Key given in Issuance is an Admin by definition.
Admin SP	The SP that has the capability to issue other SPs, and provide information about the state of SPs on the TPer as well as the TPer itself.
Authority	Defined as a Row in the Authority Table. This is a security association between an authentication Operation and a Credential, such as a public-private key pair.
Data Types	Encoding format of data. Data is encoded in different ways depending on the context in which the data is being used (stream encoding, table encoding, etc.)
Full Disk Encryption (FDE)	Data written and read to storage is encrypted before it is written and decrypted as it is read. Full Disk Encryption means that all user data through the main read-write function may be encrypted.
Platform Host	A collection of one or more Host Application resources that utilizes or provides a specific service or set of services.
Host Application	A Trusted Component (software) that initiates ATA (T13) TRUSTED SEND/RECEIVE commands or SCSI (T10) SECURITY PROTOCOL IN/OUT commands.[C D1]
IF-SEND	An interface command, such as the ATA (T13) TRUSTED SEND or SCSI (T10) SECURITY PROTOCOL IN command used to transmit data from the host to the TPer.
IF-RECV	An interface command, such as the ATA (T13) TRUSTED RECEIVE or SCSI (T10) SECURITY PROTOCOL OUT command used by the host to retrieve data from TPer, or to acquire a ComID.
Issuance	The act of activating or instantiating an SP from one or more Templates.
MAC	Message Authentication Code
Messaging	Session communications are by messages defined by a messaging protocol. Messages from a Host convey remote method calls on an SP and other messages return the results.
Method	A Method is a remote procedure call to an SP that initiates an action on the SP.
Object	Any row of an Object Table.
Personalization	The act of specializing an issued SP. Personalization requires a Transport Key from Issuance to give secure access to personalization.
PuK and PrK	Convenient notation for Public Key and Private Key.

Term	Definition
Security Subsystem Class (SSC)	For TCG Compliance and Conformance purposes a security subsystem class identifies the components from the Core specification that are Mandatory, Optional, or Excluded from a particular class of security subsystem.
Session	All communications with a specific SP. A session holds authorization state for all method invocations.
Security Provider (SP)	An atomic collection of Tables and Methods that can be issued on behalf of a host software provider.
Security Identifier (SID)	25 character passcode made up of ALPHANUMERIC CAPS, where 0 (zero) is the same as O (the letter “oh”) and 1 (one) is the same as I (the letter “eye”).
Storage Device	A Storage Device is any device that provides digital storage services.
Storage Media	The Storage Media refers to the non-volatile or persistent storage in a storage device.
Storage Working Group (SWG)	One of the TCG working groups whose purpose is to define Security building blocks for the Storage Device.
SymK	Convenient notation for symmetric key (shared secret) cryptography.
T10 Specification	SCSI CDB Specification that contains SECURITY PROTOCOL IN/OUT Commands.
T13 Specification	ATA Command Specification that contains TRUSTED SEND/RECEIVE Commands.
Table	The basic data structures within an SP. Tables store persistent SP state defined in this specification.
TPer	A Trusted Peripheral as defined by the Peripheral’s Workgroup.
Transport Key	Credential received by SP Owner during Issuance that enables the SP Owner to authenticate as the Admin authority for that SP.
Trusted Commands	Interface protocol commands (i.e. T10 SECURITY PROTOCOL IN/OUT or T13 TRUSTED SEND/RECEIVE) used to communicate with an SP.
Unique Identifier (UID)	Unique 8 byte identifier that identifies objects within tables, tables, and the SP itself.

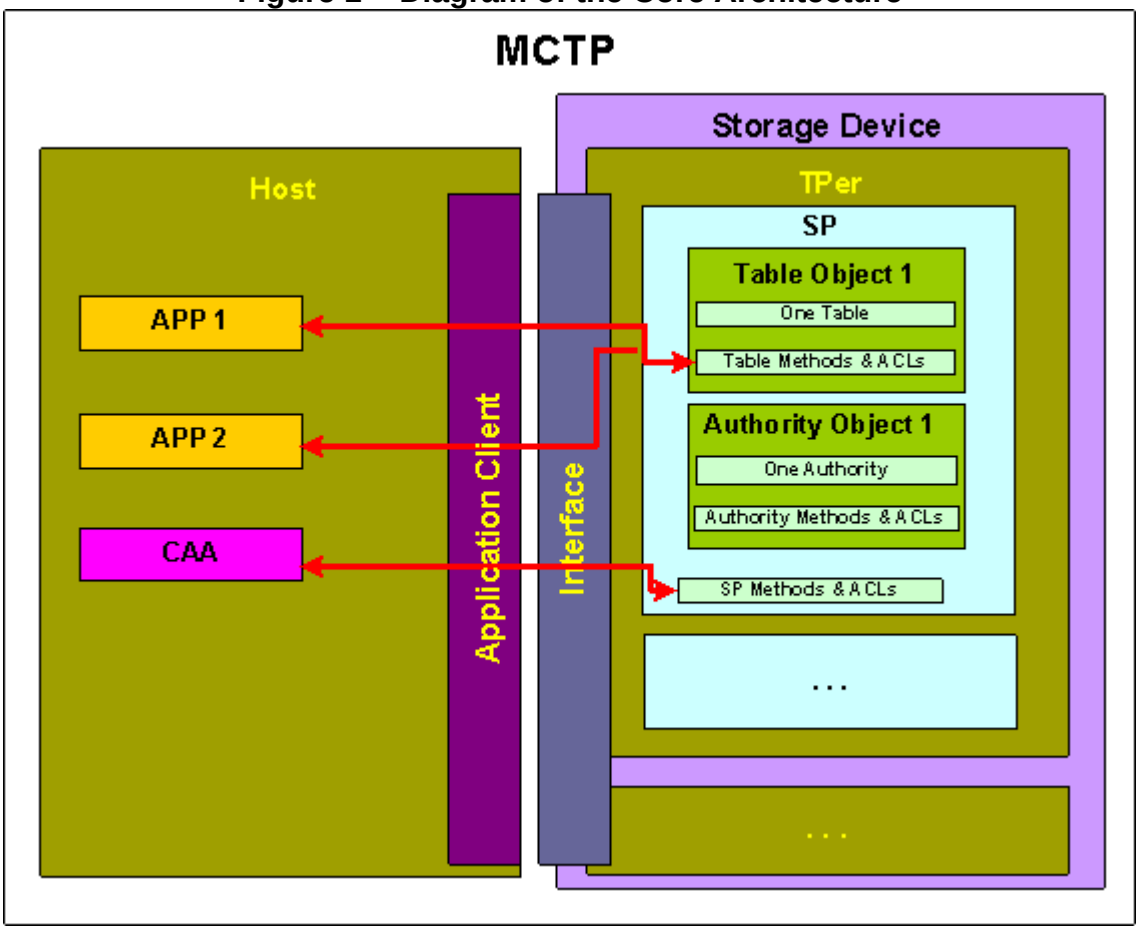
2 Trusted Storage Device Architecture

2.1 Trusted Storage Device Architecture Overview

The Core Architecture supports all of the use cases and threat models developed for the TCG Storage Use Cases. Peripherals based on this architecture are called Trusted Peripherals or TPers and reside in the storage device. This section is only a summary of the Storage Core Architecture. Refer to Section 3 and 4 for details.

2.2 Core Architecture Components

Figure 1 Diagram of the Core Architecture



The core architecture is illustrated in Figure 1 . Figure 1 shows a single Multicomponent Trusted Platform (**MCTP**) with one Trusted Peripheral (**TPer**). The MCTP supports 1 or more TPers.

2.2.1 Multicomponent Trusted Platform (MCTP)

The MCTP keeps track of the peripherals through the Component Authentication Administrator (**CAA**). Various host applications (**APPs**) communicate with the TPer using **an application client** and through **a peripheral interface** such as ATA or SCSI.

2.2.2 Host

For the purposes of this specification, a Host is the application client that initiates ATA (T13) TRUSTED SEND/RECEIVE commands or SCSI (T10) SECURITY PROTOCOL IN/OUT commands under Security

Protocol 1-6. One example of such an initiator is a PC Client as defined by the PC Client workgroup of TCG. Multiple Hosts are supported.

2.2.2.1 Host Applications (APPs) and Component Authentication Administrator (CAA)

APPs (including the CAA) can: 1) create, 2) query or 3) change the persistent state of the TPer data structures. This communication is performed using sessions. See Section 2.3.1.

2.2.3 Trusted Peripheral (TPer)

The Trusted Peripheral (TPer) resides in the Storage Device. The TPer manages trusted storage-related functions and data structures. Two main aspects to the TPer use cases as it pertains to the Core are:

1. **Data Confidentiality and Access Control over TPer features and capabilities:** TPer functions and capabilities are built upon: 1) an option for policy driven setup and 2) use of cryptographic access control over TPer content. Such features and capabilities include access-controlled readable and writeable data areas, and access-control to built-in firmware functions or hardware functions in the TPer. Furthermore, it is possible for a single trusted host application to gain exclusive access to subsets of these features and capabilities. Finally, the protection extends to confidentiality of instructions and data in transit between the trusted Host application (or a TPM it uses) and the TPer.
2. **TPers & Hosts Bilateral Enrollment and Connection: Enrollment** establishes the conditions under which data/instruction **Connections** can be established between TPers and Hosts. The access control conditions for enrollment may be different than those for connection. The data/instruction consequences of a failure to be enrolled or connected may be different for different TPers and Hosts. Finally, the permissions/authorities required for enrollment and connection of a TPer with a Host may be different than the permissions/authorities required for enrollment and connection of a Host with a TPer.

The Core Architecture provides for a system of tables where the content and meaning of the table entries may be different for different types of storage devices with different features and capabilities.

This Core Architecture's access control system scales with the available storage device resources. Storage device resources include processor performance, memory space, and media capacity. TPer data structures and operations may be fixed (and limited) or Host application-definable up to the limit of the storage device's available resources.

2.2.3.1 Security Providers (SP)

The TPer may contain one or more Security Providers (SPs).

A Security Provider (SP) is a set of tables and methods that control the persistent trust state of the SP and may participate in control of the persistent trust state of the TPer.

A Security Provider (SP) supports specific TPer functionality. Each SP has its own storage, functional scope, and security domain. SPs support functions such as authentication, secured attribute-value storage, disk encryption/decryption, backup, time stamping, and event logging. SPs are created by: 1) the manufacturer (during storage device creation) or 2) the Issuance process (see SP Issuance section 2.3.2).

A Security Provider provides a way for the Host to define: 1) which TCG functions are performed, 2) who has access to these functions, 3) how the TPer & SPs communicate with the Host, 4) when these events are permitted and 5) when the events are logged.

A Security Provider is made up of the following components:

- **Tables** are storage elements. The three table kinds are described in Section 3.2.5.1 Kinds of Tables. Tables consist of rows and columns. Tables may contain one or more rows.
- **Persistent State Information**: Table content is also known as persistent state information. This type of information remains active through power cycles, reset conditions, and spin up/down cycles. This persistent state information shall not be part of the User Addressable Logical

Block Address space on the storage device and therefore is not affected by usual partitioning or formatting of the storage device by the Host operating system.

- **Methods** are actions that are invoked on SPs, Tables, Table entries, or Objects. Method operations include functions such as: table additions, table deletion, table read access, and table backup.
- **Authorities** are authentication agents. Authorities specify passwords or cryptographic proofs required to execute the methods in the SP.
- **Access Control Lists (ACLs**) are lists of approved Boolean expressions of Authorities. ACLs bind methods to valid authorities.

2.3 Core Architecture Operations

2.3.1 Host <–> TPer Communication Infrastructure

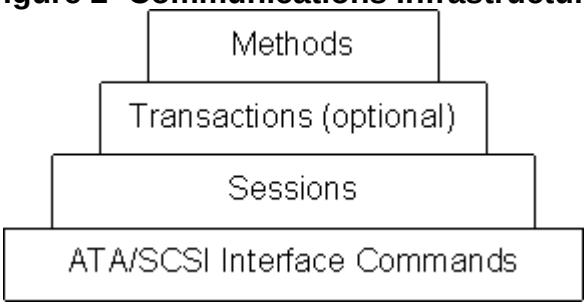
The Host communicates with SPs using Trusted commands. Trusted commands are interface-specific protocols (i.e. T10 SECURITY PROTOCOL IN/OUT Protocol 1-6, or T13 TRUSTED SEND/RECEIVE Protocol 1-6). Methods are a communications protocol transported in the payload of the interface-specific protocol.

General interface-specific protocols are defined by INCITS T10 and T13.

The SP communication protocol uses a layered communication system consisting of the following elements:

1. **Methods**: Methods are “atomic” actions that invoke SP activity.
2. **Transactions**: A transaction is a series of one or more method invocations grouped to enable state rollback to a pre-defined point if an error or abort occurs during execution of any of the methods in the series. Methods are executed either within or outside of a transaction.
3. **Sessions**: A session is a communication channel between the Host and an SP. A session requires a pair of events: 1) an “Open”, and 2) a “Close”. Transactions and method invocations occurring both inside and outside of transactions occur within sessions.

Figure 2 Communications Infrastructure



The only way to communicate with an SP is via a session. Only the host may open a session. Methods are executed within the session. Normally, when the methods and associated responses are completed, the host closes the session. Other interface-specific commands (i.e. ATA/SCSI) can be interleaved among ATA/SCSI TRUSTED/SECURITY PROTOCOL commands at any time.

A Single Session shall be Read-Only or Read-Write. If the device is capable, one or more read-only sessions may be established simultaneously to a single SP. Typically, changes made to an SP during a Read-Only session shall not persist past the end of that session. A case of a non-transient change permitted in a Read-Only session is automatic forensic logging, if enabled.

Read-Write Sessions may or may not alter persistent state information (table content). A Read-Write session (one which has the capability of making non-transient changes to an SP) shall be unable to run simultaneously with any other sessions to the same SP.

Secure Messaging provides session communications that support message confidentiality, message integrity/authenticity, or both. Using previously-defined security attributes, the host and TPer may pass encrypted or integrity protected messages (methods and their associated responses) during sessions. Message encryption is recommended but not required. When secure messaging is in use, it is done regardless of and in addition to any encryption done on the communications channel.

TPer-Host Attachment/Communication. In the simplest case the host is just the platform host to which the TPer is directly attached or attached over a network. The host could be some other platform host that communicates with the immediate platform host, which relays the session stream to the TPer over a network. In another case, the TPer could be wirelessly connected to its host, or part of a SAN and connected to multiple hosts.

2.3.2 SP Issuance & Personalization Overview

New SPs are created or modified using sessions and methods.

Issuance is the act of creating a new SP. When TPers are capable of SP issuance, special resources called Templates are required. **Templates** define the initial tables and methods upon which new SPs are based when issued. All SPs incorporate the Base Template's tables and methods. Other Templates are combined with the Base Template to extend its functionality. Some Templates that may extend the Base Template are: Admin Template, Clock Template, Crypto Template, Locking Template, and Log Template.

Personalization is the customization of a newly created SP. The primary purposes of Personalization are modification of the SP's initial table data and/or the administrative authority on that specific SP, as well as customization of the default access control settings.

2.3.3 Security Subsystem Classes Overview

The Core Specification defines all possible TCG-related functions supported by a TPer. However, every TPer is not required to support all functionality defined in this specification. There shall be multiple "classes" of Core Specification compliance, called Security Subsystem Classes (or SSCs). Each Security Subsystem Class specification is a companion document to the Core Specification.

Security Subsystem Classes explicitly define the minimum acceptable Core Specification capabilities of a TPer in a specific "class". A TPer in a specific class may have only some of the capabilities (tables, methods, access controls) defined in this Core Specification and may include additional capabilities through table definitions. No Security Subsystem Class shall replace a capability called out in the Core Specification with the same capability implemented in different tables, methods, and access controls.

Security Subsystem Classes define only TCG-related functionality. TPer attributes such as host interface type, storage capacity, data rates, and seek times are not key Security Subsystem Class attributes, though TPer resources such as available memory, storage capacity, and processing power influence which Security Subsystem Class(es) a TPer supports.

3 Core Architecture Elements

3.1 Core Architecture Elements Overview

This section defines global TCG storage-related document format, data structures, and functional behavior.

3.2 Data Structure Descriptions

3.2.1 Document Data Formats

This specification defines three distinct but closely related data models:

- **Tables:** Data stored in tables is of a maximum fixed size. When a table is created it is always allocated with a fixed number of fixed-size columns. Some Security Subsystem Classes may require that tables be created with a pre-allocated maximum number of rows.
- **Messaging**: Data moving across the interface is encoded into byte streams. These streams carry encodings for method calls, parameters, and results, as well as some other control information. There are no predefined limits on the size or length of these streams, but the TPer may limit the maximum size of encoded values.
- **Exposition Pseudo-code**: This provides a C-like definition of Methods and Table contents. The definition of the exposition pseudo-code is in next section 3.2.2.1.

3.2.1.1 Tables – Example

For the example in the following sections, a table named “Foo” is used. This specification documents the “Foo” table in the following manner:

Table 03 Foo Table Description

Column	Type	Description
ID	uid	UID of the entry.
Username	name	Name of the user.
SerialNumber	uinteger_4	Serial Number of item purchased.

3.2.1.2 Methods – Example

The value of row 2, column “Username” of table “Foo” will be set to “Alice”.

In the pseudo-code, the Method invocation is shown as:

```
Foo.Set[ [ startRow=2, startColumn=”Username” ], “Alice” ]
=>
[boolean]


“=>” is the separator between the method call specification and the return result specification.
```

Note that since actual method invocation is performed using UIDs and not names, “Foo.Set” would be replaced by the UIDs for “Foo” and “Set.”

3.2.2 Data Types

Data is encoded in different ways depending on the context in which the data is being used. One data context is data stored in tables. Another data context is data crossing the interface in messaging – this is called “Stream Encoding”.

This section introduces the different data types, provides a brief introduction on how these types are used, and shows how they are displayed in this document. See Section 3.2.3 for additional details regarding data types and data type Stream Encoding.

3.2.2.1 Pseudo-code (Expository)

Pseudo-code is used to describe types, method parameters, and snippets of code without having to use the byte encodings directly.

Method parameters are of two kinds: required and optional. Required parameters must come in the order defined in this specification, must precede the optional parameters, and are not named values. Optional parameters are passed as named values. Optional parameters are not required to be in order, and are not required to be included in a method invocation.

In the pseudo-code, the required parameters are given expositional names for ease of reference, and are formatted as: `Expositional-Name : Parameter-type.` Optional parameters are given in the form of Named values, except the right hand side in the prototype is the type of the value, as in: `tableName = Parameter-type`.

A method's result shall be contained in a list, and shall be followed by an End of Data token and a status code list. Note that the return value(s) are annotated with the same convention as parameters, and may contain optional parts that are also passed as named values. The result list of a failed method invocation should be empty.

In this document

SP method calls are written as:	`SPUID.MethodName[ <Parameters> ]`
Table method calls are written as:	`TableNameUID.MethodName[ <Parameters> ]`
Object methods are written as:	`ObjectNameUID.MethodName[ <Parameters> ]`

For example:

Calling an SP method:	`SPUID.Random[ <Parameters> ]`
Adding an entry to a log table:	`SomeLogTableUID.AddLog[ <Parameters> ]`
Encrypting host data:	`C_AES_128ObjectUID.Encrypt[ <Parameters> ]`

`SPUID` always represents the UID reserved to refer to “this SP.” This reserved UID is `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01`. TableNameUID represents the UID of a particular table. In the case of this example, the UID would be of the table named “TableName”. ObjectNameUID represents the UID of a particular object. In the case of this example, the UID would be of the object named “ObjectName” (see 3.2.5.3 for information on UIDs).

3.2.2.2 Messaging Data Types

There are two data types defined specifically for messaging.

- **Named values**. Named values are used to send the optional parameters in method calls.
- **List values**. List values are used to encode method parameter lists and return results.

Named values identify method parameters in the stream encoding. List values are used to group method parameters or to separate the method signature from the method parameters in the stream encoding. For more information on stream encoding, see 3.2.3.

3.2.2.3 Method Parameter/Column Value Typing and Encoding

All method parameter values and all column values are defined by types that appear in the `Type` table of the Base Template (see Table 37). Each type has a UID associated with it. Types may be derived from combinations of other types.

Types are divided into two categories: base types and derived types.

Base types are the types from which all other types are built. The base types are:

- integer – a signed mathematical integer
- uinteger – an unsigned mathematical integer
- bytes
- bytes {max=n}

Derived types are built from other types in combination with a format identifier. The format identifier is used to indicate the way in which the types are to be combined. A simple type is a type that is derived directly from a base type. For more information on types, see 5.1.

To enable the TPer to identify the type used for each method parameter and correctly store that value in a table, if necessary, and to allow the TPer to type check method parameter values, each method parameter, when it is transmitted across the interface, is prefixed with the UID of the type of that parameter.

In order to encode a parameter that is a simple, enumeration, reference, or name-value type, the UID of that type is encoded, followed by the encoded parameter value.

For parameters that are list types, the type for the list need be encoded only once, and the list itself then enclosed with the Start List and End List tokens (see 3.2.3 for token information), rather than encoding the list type for each element of the list.

To encode a parameter that is an alternative type, the UID of that alternative type is encoded, followed by the encoded UID of the type that was selected from among the alternatives, followed by the encoded parameter value.

To encode a parameter that is a struct type, the UID of that struct type is encoded, followed by the encoded type UID and value for each of the struct's included components – the group of UIDs and values of the struct's components shall be enclosed with the Start List and End List tokens.

To encode a parameter that is a set type, the UID of the set type is encoded, followed by the encoded typ UID and value for each of the set's type – the group of UIDs and values for the set's components shall be enclosed with the Start List and End List tokens.

For an example of encoding using this mechanism, see 3.2.3.3.2.

3.2.3 Stream Encoding

The messaging model provides for stream encoding of multiple remote procedure calls and multiple responses in the same interface command payloads, with the purpose of permitting large data blocks to be broken up and submitted in parts, for the parts to be acted on, and for the results to be returned in parts. This streaming model permits results to be asynchronously returned before all the parts are received.

This section details how values and control markers are encoded into byte sequences for transport over session streams (byte streams).

3.2.3.1 Data Types

As introduced in Section 3.2.2, data is encoded using four basic types of values. These four types can represent all of the basic and derived data types.

- **Integers:** Integer values are used to represent numbers, Booleans, and enumerations. In the interface (session stream) and in tables they are big endian. The implementation is free to use other representations in other circumstances, converting as necessary.
- **Bytes:** These are sequences of 8 bit bytes and are used to represent strings, cryptographic keys, bit-vector encoded sets, blobs, etc.
- **List:** Zero or more values of any type, grouped into an ordered list ([3, "abc", false])**.**
- **Named:** The name (a byte-value) followed by its value (any messaging type). A named value attaches a name to some other value (size=32).

3.2.3.2 Tokens

Values of the four basic types are packaged into tokens, each of which is a (tag, length, value) sequence of bits that specify a single data value.

Table 04 Token Types

Byte 0	Byte 1	Byte 2	Byte 3	Hex	Acronym	Meaning
0 S d<5..0>				00..7F		Tiny atom
1 0 B S n<3..0>				80..BF		Short atom
1 1 0 B S n<10..0>				C0..DF		Medium atom
1 1 1 0 0 0 B S	n<23..16>	n<15..8>	n<7..0>	E0..E3		Long atom
				E4..EF	TCG Reserved	
1 1 1 1 0 0 0 0				F0	SL	Start List
1 1 1 1 0 0 0 1				F1	EL	End List
1 1 1 1 0 0 1 0				F2	SN	Start Name
1 1 1 1 0 0 1 1				F3	EN	End Name
				F4..F7	TCG Reserved	
1 1 1 1 1 0 0 0				F8	CALL	Call
1 1 1 1 1 0 0 1				F9	EOD	End of Data
1 1 1 1 1 0 1 0				FA	EOS	End of session
1 1 1 1 1 0 1 1				FB	ST	Start transaction
1 1 1 1 1 1 0 0				FC	ET	End of transaction
				FD..FF	TCG Reserved	

The Token Types identified in Table 04 are divided into 3 subgroups:

- Simple Tokens - Atoms: tiny, short, medium, and long atoms
- Token Sequences: Start List, End List, Start Name, and End Name
- Control Tokens: Call, End of Data, End of Session, Start Transaction, End Transaction

Additionally, tokens `0xE4-0xEF`, `0xF4-0xF7` and `0xFD-0xFF` are reserved for use by TCG.

3.2.3.2.1 Simple Tokens – Atoms Overview

Atoms can be tiny atoms, which are one byte in length; short atoms which have a 1-byte header and can contain up to 15 bytes of data; medium atoms which have a 2-byte header and can contain up to 2047 bytes of data; and long atoms which have a 4-byte header and which can contain up to 16,777,215 bytes of data.

Integer values should be encoded using the shortest possible atom. Tiny atoms only represent integers, whereas short, medium, and long atoms can be used to represent integers or bytes (with the “B” bit set).

A continued value is used to represent a long byte sequence when the total length is not known in advance. A continued value is represented by a sequence of two or more atoms. Each atom may be a short atom, medium atom, or long atom. The BS bits are set to 11b for all atoms except the last atom, for which the BS bits are set to 10b. All representations of continued values are considered equivalent

encodings of the same value. Thus a 100-byte value could be split up into ten 10-byte atoms; two 50-byte atoms; or two 25-byte atoms, four 10-byte atoms, an 8-byte atom, and a 2-byte atom.

3.2.3.2.1.1 Tiny atoms

Tiny atom header and data are all contained in eight bits.

Table 05 Tiny Atom Description

Header+Data							
Tiny atom	sign	data					
0	S	*d*	*d*	*d*	*d*	*d*	*d*

The encoding is as follows:

Table 06 Tiny Atom Encoding

Tiny Atom indicator	This bit is set to 0b to indicate the atom is a tiny atom
Sign indicator	***Value*** ***Interpretation*** 0b The data is treated as unsigned integer data. 1b The data is treated as a signed integer.
Data bits	These represent the data value, an unsigned value in the range of 0...63 or a signed value in the range of –32...31. The interpretation will be based on the setting of the sign bit.

3.2.3.2.1.2 Short atoms

Short atoms consist of a one-byte header and between 1 and 15 bytes of data.

Table 07 Short Atom Description

Header (1 byte)								Data	
Short Atom		byte/ integer	sign/ continued	length				(1...15 bytes)	
1	0	B	S	*n*	*n*	*n*	*n*	*d*	...	*d*

The encoding is as follows:

Table 08 Short Atom Encoding

Short Atom indicator	These two bits are set to 10b to indicate the atom is a short atom.
Byte/integer indicator	***Value*** ***Interpretation*** 0b The data bytes represent an integer value and the S bit indicates if that value is signed. 1b The data bytes represent a byte sequence and the S bit indicates whether or not this value is continued into another atom.

Sign/continued indicator	***Value*** ***Interpretation*** 0b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as unsigned integer data. B==1b The data is either the complete byte sequence, or the final segment of a continued byte sequence. 1b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as signed integer data. B==1b The data is a non-final segment of a multi-byte continued value.
Length	These bits specify the length of the following data byte sequence. The value 0 is not a legal value. The permitted range is up to 15.

3.2.3.2.1.3 Medium atoms

Medium atoms consist of a two-byte header, and between 1 and 2047 bytes of data.

Table 09 Medium Atom Description

Header (2 bytes)													Data		
0								1					...		
Medium Atom			byte/ integer	sign/ continued	length								(1..2047 bytes)		
1	1	0	B	S	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*d*	...	*d*

The encoding is as follows:

Table 10 Medium Atom Encoding

Medium Atom indicator	These three bits are set to 110b to indicate the atom is a medium atom.
Byte/integer indicator	***Value*** ***Interpretation*** 0b The data bytes represent an integer value and the S bit indicates if that value is signed. 1b The data bytes represent a byte sequence and the S bit indicates whether or not this value is continued into another atom.
Sign/continued indicator	***Value*** ***Interpretation*** 0b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as unsigned integer data. B==1b The data is either the complete byte sequence, or the final segment of a continued byte sequence. 1b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as signed integer data. B==1b The data is a non-final segment of a multi-byte continued value.
Length	These bits specify the length of the following data byte sequence. The value 0 is not a legal value. The permitted range is up to 2047.

3.2.3.2.1.4 Long atoms

Long atoms consist of a four-byte header, and between 1 and 16M bytes of data.

Table 11 Long Atom Description

Header (4 bytes)																										Data		
0								1				2				3										...		
Long Atom				reserved		byte/ integer		sign/ continued		Length																(1..16,777,215 bytes)		
1	1	1	0	0	0	B		S		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*d*	...	*d*

The encoding is as follows:

Table 12 Long Atom Encoding

Long Atom indicator	These four bits are set to 1110b to indicate the atom is a long atom.
reserved	These bits are reserved and shall be set to 0b.
Byte/integer indicator	***Value*** ***Interpretation*** 0b The data bytes represent an integer value and the S bit indicates if that value is signed. 1b The data bytes represent a byte sequence and the S bit indicates whether or not this value is continued into another atom.
Sign/continued indicator	***Value*** ***Interpretation*** 0b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as unsigned integer data. B==1b The data is either the complete byte sequence, or the final segment of a continued byte sequence. 1b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as signed integer data. B==1b The data is a non-final segment of a multi-byte continued value.
Length	These bits specify the length of the following data byte sequence. The value 0 is not a legal value. The permitted range is up to 16,777,215.

3.2.3.2.2 Encoding Example

An example encoding of a medium atom can be found in Table 13. The bit organization in the header of the example in Table 13 can be found in Table 14.

Table 13 Medium Atom Encoding Example

D0 1C	54	48	49	53	49	53	41	4E	45	58	41	4D	50	4C	45	4F	46	41	4D	45	44	49	55	4D	41	54	4F	4D
Medium Atom Header	T	H	I	S	I	S	A	N	E	X	A	M	P	L	E	O	F	A	M	E	D	I	U	M	A	T	O	M

Table 14 Medium Atom Header Encoding Example

Byte 1					Byte 2										
medium atom			byte/integer	sign/continued	length										
1	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0
D0					1C										

3.2.3.2.3 Token Sequences

Composite values are represented by a sequence of tokens.

3.2.3.2.3.1 Named

Named values have the form `name=value` and are used to represent an attribute-value pair. A named value is a sequence of tokens: a `Start Name` token (SN), followed by a non-continued byte value that specifies the name, followed by any value (including list or Named values), followed by an `End Name` token (EN token).

For example, the named value `foo=3` would be encoded may be encoded using a short atom for "foo" and a tiny atom for 3 as shown in Table 15.

Table 15 Named Value Encoding Example

F2	A3	66	6F	6F	03	F3
SN	short atom	"f"	"o"	"o"	3	EN
	Name				*value*	

Note that the value is not constrained to be a single integer as shown here, but could be anything, including another named value or a sequence. Table 16 shows an encoding of "foo=bar=3" using a short atom for "foo", a short atom for "bar", and a tiny atom for 3.

Table 16 Named Value/Sequence Encoding Example

F2	A3	66	6F	6F	F2	A3	62	61	72	03	F3	F3
SN	short atom	"f"	"o"	"o"	SN	short atom	"b"	"a"	"r"	3	EN	EN
	Name					*name*				*value*		
					Value							

3.2.3.2.4 List

Lists are ordered sequences of elements of the form `[e1,e2,...,ei]`. List elements may be tokens, token lists, or named tokens. A list is encoded as a `Start List` token (SL) followed by a sequence of zero or more elements followed by an `End List` token (EL).

For example, the sequence `[3, 4, [5, 6]]` is a sequence of three tiny atoms encoded as shown in Table 17.

Table 17 List Value Encoding

F0	03	04	F0	05	06	F1	F1
SL	3	4	SL	5	6	EL	EL

3.2.3.2.5 Control Tokens

Control tokens are single byte tokens that are used to specify special actions.

- **Call**. Used to start a method call.
- **End of Data**. Used to signal the end of the parameters, or the result, of a method call. This is used in message streams by both the host and the SP.
- **End of Session**. Used to end a session.
- **Start Transaction**. Used to open a transaction. When the host begins a transaction, the Start Transaction token is sent by the host to the SP and is immediately followed by the status required for that transaction control token. When the SP delivers its response, its message shall mirror that of the host by including Start Transaction tokens in the equivalent places in the message stream, along with the actual status of the Start Transaction request. See example encoding in 3.2.3.3.3.
- **End Transaction**. Used to commit or abort a transaction. When the host ends a transaction, the End Transaction token is sent by the host to the SP and is immediately followed by the status required for that transaction control token. When the SP delivers its response, its message shall mirror that of the host by including End Transaction tokens in the equivalent places in the message stream, along with the actual status of the End Transaction request. Sending the End Transaction token and a status code of `0x01` aborts a transaction. See example encoding in 3.2.3.3.3.

In cases where the host transmits unexpected or out of order control tokens the TPer should abort the session. These cases include (but are not limited to):

- Multiple consecutive tokens of the same type
- Out of order tokens
- Tokens with undefined accompanying status codes

3.2.3.3 Method Calls

This section describes the encoding of method calls.

3.2.3.3.1 Syntax

A method call starts with a sequence of tokens that are sent from the application to the TPer as follows:

- **Method.** A call token followed by tokens for a value that identifies the method to call. This value is:
    - **InvokingUID, MethodUID –** This indicates the method invocation is a series of tokens with two short atom elements. InvokingUID here is the UID of the table, object, or “this SP” upon which the method is being invoked, and MethodUID is the UID of the method as recorded in the `MethodID` table. The InvokingUID of an SP invoking a method shall always be `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01`, which is used to signify “this SP”.

      Session Manager layer methods follow this format as well. The host shall use the reserved UID "SMUID" (`0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF`) as the InvokingUID of Session Manager methods.
- **Input Parameters.** This is a token list of the method invocation’s parameters. Positional parameters, those required by the method invocation, shall appear first, in the order in which

they are listed in this specification for that method, and are not named. Optional parameters shall appear after all positional parameters. Optional parameters are named values. These parameters may be listed in any order, but shall follow the list of all positional parameters for that method.

Positional parameters shall be made up of the following parts:

- The UID of the row in the `Type` table that represents the value's type.
  - For certain types, the UID of the component value
- The appropriate Atom identifier for the value.
- The encoded value.

Optional parameters shall be made up of the following parts:

- The StartName token.
- The appropriate Atom identifier for the name.
- The encoded name.
- The UID of the row in the `Type` table that represents the value's type.
  - For certain types, the UID of the component value
- The appropriate Atom identifier for the value.
- The encoded value.
- The EndName token.

- **EndOfData.** An end of data token sequence.
- **Status Code List.** This is the status list that contains the status code expected from successful invocation of the method. The first value in the list shall be `0x00` for a method that the host expects to complete properly. For a method that the host wishes to abort, the host shall include a value that is not `0x00` as the first value in the status list, which shall cause the TPer to abort processing on that method and return that non-`0x00` value as the first value in the status list.

  The second and third values in the status list are reserved, and are defined in this specification to be zeroes.

Each method call shall have a response that is a sequence of tokens that are sent from the TPer to the host as follows.

- **Output Results.** This is a token list.
- **EndOfData.** An end of data token sequence.
- **Status List.** The status list returned for the method invocation. The first value in the status list shall always be the status of the method, as described in 5.1.3. The second and third values in the list are uintegers reserved for use by the TCG, and are defined in this specification to be zeroes.

  Additional values may be returned in the status list, as long as the first three values in the status list are returned as required by this specification.

Method responses shall be returned for all method invocations or method invocation attempts within a session, though responses for method invocation attempts of methods not recognized by the TPer or that result in some other failure condition shall return an empty method result (the output result is an empty list) and an error code. Unrecognized method invocation attempts outside of a session should be ignored by the TPer – in these cases, no response is sent.

Session Manager protocol layer method invocations that are recognized but fail shall receive the normal response format for that method, accompanied by an error status code. Session startup methods that fail in this way shall have returned the expected method response, but that method shall have only the identifying parameters (Host, SP) and an error status code. If the identifying parameters (particularly the Host parameter) are invalid (i.e. of the incorrect type), the TPer may ignore the method.

All message traffic to invalid/non-existent streams and/or sessions shall be ignored by the TPer.

The TPer may begin sending the response as soon as enough parameters have been received to prepare a response.

3.2.3.3.2 Method Call Encoding Stream – Example

The method example in this section invokes the **`CreateRow`** method on the `Authority` table using the following byte stream format. The encoded stream data for the example method invocation is can be found shown in Table 18. For the result of the method as returned by the TPer, see Table 19.

Table 18 Method Call Encoding

Call	token	Authority	token	CreateRow
F8	A8	00 00 00 09 00 00 00 00	A8	00 00 00 06 00 00 00 04

[	SN	token	Name	token	name	token	Alice	EN
F0	F2	A4	4E 61 6D 65	A8	00 00 00 05 00 00 02 0B	A5	41 6C 69 63 65	F3

SN	token	CommonName	token	name
F2	AA	43 6F 6D 6D 6F 6E 4E 61 6D 65	A8	00 00 00 05 00 00 02 0B

token	AliceGroup	EN
AA	41 6C 69 63 65 47 72 6F 75 70	F3

SN	token	IsClass	token	boolean	F	EN
F2	A7	49 73 43 6C 61 73 73	A8	00 00 00 05 00 00 04 01	0	F3

SN	token	Enabled	token	boolean_def_true	T	EN
F2	A7	45 6E 61 62 6C 65 64	A8	00 00 00 05 00 00 04 03	1	F3

SN	token	Secure	token	messaging_type	0	EN
F2	A6	53 65 63 75 72 65	A8	00 00 00 05 00 00 04 04	0	F3

SN	token	HashAndSign	token	hash_protocol	0	EN

SN	token	HashAndSign	token	hash_protocol	0	EN
F2	AB	48 61 73 68 41 6E 64 53 69 67 6E	A8	00 00 00 05 00 00 04 0D	0	F3

SN	token	Operation	token	auth_method	1	EN
F2	A9	4F 70 65 72 61 74 69 6F 6E	A8	00 00 00 05 00 00 04 08	1	F3

SN	token	Credential	token	cred_object_uidref	token	PinObjectUID	EN
F2	AA	43 72 65 64 65 6E 74 69 61 6C	A8	00 00 00 05 00 00 10 02	A8	00 00 00 0B 00 00 00 09	F3

]	EOD	[	Status			]
F1	F9	F0	00	00	00	F1

Which returns, upon success with:

[true]

Table 19 Method Response Encoding

[	createrow_result	uidref_list	[	New Authority Object	]	]
F0	00 00 00 05 00 00 06 07	00 00 00 05 00 00 08 09	F0	00 00 00 09 00 FF FF 01	F1	F1

EOD	[	Status			]
F9	F0	00	00	00	F1

3.2.3.3.3 Method Encoding with Transactions – Example

This example displays the encoding of a series of methods that utilize transactions. The method invocation is in Table 20. The response from the TPer is encoded in Table 21.

Table 20 Method Call Encoding with Transaction

Begin Transaction	Status	Call	token	Authority	token	CreateRow
FB	00	F8	A8	00 00 00 09 00 00 00 00	A8	00 00 00 06 00 00 00 04

[	SN	token	Name	token	name	token	Alice	EN
F0	F2	A4	4E 61 6D 65	A8	00 00 00 05 00 00 02 0B	A5	41 6C 69 63 65	F3

SN	token	CommonName	token	name
F2	AA	43 6F 6D 6D 6F 6E 4E 61 6D 65	A8	00 00 00 05 00 00 02 0B

token	AliceGroup	EN
AA	41 6C 69 63 65 47 72 6F 75 70	F3

SN	token	IsClass	token	boolean	F	EN
F2	A7	49 73 43 6C 61 73 73	A8	00 00 00 05 00 00 04 01	0	F3

SN	token	Enabled	token	boolean_def_true	T	EN
F2	A7	45 6E 61 62 6C 65 64	A8	00 00 00 05 00 00 04 03	1	F3

SN	token	Secure	token	messaging_type	0	EN
F2	A6	53 65 63 75 72 65	A8	00 00 00 05 00 00 04 04	0	F3

SN	token	HashAndSign	token	hash_protocol	0	EN
F2	AB	48 61 73 68 41 6E 64 53 69 67 6E	A8	00 00 00 05 00 00 04 0D	0	F3

SN	token	Operation	token	auth_method	1	EN
F2	A9	4F 70 65 72 61 74 69 6F 6E	A8	00 00 00 05 00 00 04 08	1	F3

SN	token	Credential	token	cred_object_uidref	token	PinObjectUID	EN
F2	AA	43 72 65 64 65 6E 74 69 61 6C	A8	00 00 00 05 00 00 10 02	A8	00 00 00 0B 00 00 00 09	F3

]	EOD	[	Status			]	EndTransaction	Status
F1	F9	F0	00	00	00	F1	FC	00

Table 21 Method Response Encoding – with Transaction

Begin Transaction	[	createrow_result	uidref_list	[	New Authority Object
FB	F0	00 00 00 05 00 00 06 07	00 00 00 05 00 00 08 09	F0	00 00 00 09 00 FF FF 01

]	]	EOD	[	Status			]	End Transaction	Status
F1	F1	F9	F0	00	00	00	F1	FC	00

3.2.3.4 ComPackets, Packets & Subpackets

Data crosses the Host/TPer interface in T10 SECURITY PROTOCOL IN/OUT or T13 TRUSTED SEND/RECEIVE commands.

The low-level interface transport layer shall handle the retransmission of damaged or incomplete commands. Secure messaging, detailed in later sections of this specification, permits the host

application to secure its data from malicious attack, not to address hardware and low-level transport issues. (Similarly with the session start up protocol, hashing is intended to detect tampering.)

The payloads of the interface commands convey tokenized byte streams (method calls, their parameters, their results, and status codes) and other control information, such as ACKs and NAKs.

3.2.3.4.1 Format

There are three levels of packetization on the host interface: **ComPackets, Packets, and Subpackets**. A **ComPacket** is the unit of communication transmitted as the payload of an interface command. An interface command payload shall hold only one ComPacket. A ComPacket shall not span multiple interface commands. A ComPacket is able to hold multiple packets in its payload. A **Packet** is associated with a particular session and may hold multiple subpackets. A **Subpacket** may hold multiple Tokens. Tokens may span multiple subpackets and multiple packets. However, subpackets cannot span multiple packets, and packets cannot span multiple ComPackets.

Figure 3 provides an overview of how ComPackets are constructed from packets; how packets are constructed from subpackets; and how subpackets are constructed from the session byte stream.

Figure 3 Packet Construction

Session Byte Stream			
Already Sent	Being Sent in First Subpacket	Being Sent in Second Subpacket	Pending
B_0 B_1 B_2 B_3 B_4 B_5	B_6 B_7 B_8 B_9 B_{10} B_{11}	B_{12} B_{13} B_{14} B_{15} B_{16} B_{17} B_{18} B_{19}	B_{20} B_{21} B_{22} B_{23} B_{24} …

First Subpacket			
Kind	Reserved	Length	Data
0 0	0 0 0 0	0 0 0 6	B_6 B_7 B_8 B_9 B_{10} B_{11}

Second Subpacket			
Kind	Reserved	Length	Data
0 0	0 0 0 0	0 0 0 8	B_{12} B_{13} B_{14} B_{15} B_{16} B_{17} B_{18} B_{19}

Packet					
Session Number	SeqNumber	AckType	Acknowledgement	Length	Data
0 0 0 1 0 0 0 1	0 0 0 1	0 0	0 0 0 0	0 0 0 22	0 0 0 0 0 0 0 0 0 6 B_6 B_7 B_8
Data					
B_9 B_{10} B_{11} 0 0 0 0 0 0 0 0 0 8 B_{12} B_{13} B_{14} B_{15} B_{16} B_{17} B_{18} B_{19}					

Com Packet					
Reserved	Extended ComID	OutstandingData	MinTransfer	Length	Data
0 0 0 0	Com Com Com Com	0 0 0 5	0 0 0 5	0 0 0 38	0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0
Data					
0 0 0 22 0 0 0 0 0 0 0 0 0 6 B_6 B_7 B_8 B_9 B_{10} B_{11} 0 0 0 0 0 0 0 0 0 8 B_{12} B_{13} B_{14} B_{15} B_{16} B_{17} B_{18} B_{19}					

3.2.3.4.2 ComPacket Format

- **Header**
  - **Reserved:uinteger_4 –** must be all zeros.
  - **ExtendedComID:uinteger_4 –** The ComID of this ComPacket
  - **OutstandingData:uinteger_4 –** For ComPackets sent by the TPer to the Host, this field contains the total number of bytes that the TPer has available for the host on this ComID. This value is based on the data available in the TPer at the point in time when the ComPacket is transmitted to the host by the TPer. This total shall not include the data

being transferred in the current ComPacket. This total shall include Compacket/Packet/Subpacket overhead. If the TPer has no additional data for this ComID, this value shall be 0x00 0x00 0x00 0x00. If the TPer has more than 0xFF 0xFF 0xFF 0xFF bytes for this ComID, this value shall be 0xFF 0xFF 0xFF 0xFF. For ComPackets sent by the Host to the TPer, this field is reserved and shall contain 0x00 0x00 0x00 0x00.

- **MinTransfer:uinteger_4 –** For ComPackets sent by the TPer to the Host, this field contains the minimum number of bytes that the host must request on this ComID in order to transfer a packet for any session associated with this ComID. This value is based on the data available in the TPer at the point in time when the ComPacket is sent by the TPer. This value shall include Compacket/Packet/Subpacket overhead. If the TPer has no additional data for this ComID, or if the TPer has no minimum requirement, this value shall be 0x00 0x00 0x00 0x00. The host application that manages this ComID shall request at least MinTransfer bytes on the next IF-RECV command that it sends for this ComID. For ComPackets sent by the Host to the TPer, this field is reserved and shall contain 0x00 0x00 0x00 0x00.
- **Length:uinteger_4 –** The number of bytes in the payload

- **Payload**
  - **Data:bytes{length}**. This contains a sequence of one or more packets.

3.2.3.4.3 Packet Format

Each packet will be made up of the fixed fields noted below, to allow acknowledgements, negative acknowledgements, and/or data to be included in a single packet.

- **Header**
  - **Session: uinteger_8** – The session number associated with this packet. The session number is composed of two uinteger_4 values – the TPer session number and the Host session number (Session = TPerSN concatenated with the HostSN). The TPer Session Number is sent first; the Host Session Number is second. Consequently, the same session number is used for communications between both parties.
  - **SeqNumber: uinteger_4 –** An incrementing counter that starts at 1 and increments until 2^{32}-1, which identifies the number of the packet within the session and defines the ordering of transmitted packets.

    The message recipient shall ignore a packet with an equal or lower SeqNumber value than any previously acted-upon packet. In addition, wrapping of the SeqNumber shall result in the session being automatically aborted.

    Each communicator shall maintain multiple SeqNumber counts, including that of the last packet acknowledged, the next packet expected, and the last packet transmitted.
  - **AckType: uinteger_2 –** This will be 0x00 0x01 if the Acknowledgement field is to contain a packet acknowledgement. This will be 0x00 0x02 if the Acknowledgement field is to contain a packet negative acknowledgement. This will be 0x00 0x00 if no packets are being acknowledged or negative acknowledged, and the value of the Acknowledgement field shall be zeroes.
  - **Acknowledgement: uinteger_4 –** If the value of the AckType field is 0x00 0x01, then this number shall be the SeqNumber of the last packet successfully received by the receiver. If the value of the AckType field is 0x00 0x02, then this shall be the SeqNumber of the packet at which the receiver wishes the sender to begin retransmission. Generally, the receiver will put a value of the last known good packet received plus one. For AckType field value of 0x00 0x02, the communicator shall not NAK a SeqNumber less than or equal to the last ACKed SeqNumber. If the AckType field is 0x00 0x00, then the value of this field shall be zeroes.
  - **Length: uinteger_4** – The number of bytes in the Payload field.

- **Payload**
  - **Data: bytes{length}** – This contains a sequence of one or more subpackets.

3.2.3.4.4 Data Subpacket Format

A Data Subpacket consists of the following fields:

- **Header**
  - **Kind: uinteger_2 –** This field is set to zeroes to identify this as a data subpacket.
  - **Reserved: uinteger_4 –** These bytes are reserved. This specification requires these bytes to be zeroes.
  - **Length: uinteger_4** – The number of bytes in the subpacket payload. This is equal to the number of bytes in the subpacket payload.
- **Payload**
  - **Data: bytes{length}** – This contains a series of bytes representing one, more than one, or perhaps part of one token.

3.2.3.4.5 Credit Control Subpacket Format

A Credit Control Subpacket consists of the following fields. For more information on the use of Credit Control Subpackets, see Flow Control in Section 3.4.6.

- **Header**
  - **Kind: uinteger_2**. This is `0x80 0x01`, and identifies this subpacket as a credit control subpacket.
  - **Reserved: uinteger_4** – These bytes are reserved. This specification requires these bytes to be zeroes.
  - **Length: uinteger_4**. The number of bytes in the Credit Control subpacket payload. This is always `0x00 0x00 0x00 0x02` for a subpacket of this type.
- **Payload**
  - **Credit: uinteger_2**. The number of bytes to credit. It's an additional number of bytes that may be sent to the stream.

3.2.3.5 Secure Messaging

Secure messaging enables protection of the packet payload. Secure messaging comes in three types:

- Confidential Messaging – this provides encryption on the message being transmitted. Confidential Messaging prevents the packet contents from being read by an intruder between the packet source and destination.
- Integrity/Authenticity Checking – this provides the ability to detect tampering with packets in a session.
- Confidential Messaging with Integrity/Authenticity Checking – this provides encryption on the message being transmitted and the added ability to detect tampering with packets in a session.

3.2.3.5.1 Secure Messaging Packet Format

A secure messaging packet is used when encryption or integrity/authenticity checking (or both) is enabled for a session. The secure messaging packet is composed of the following fields:

- **Header**
  - **Session: uinteger_8** – The session number associated with this packet. The session number is composed of two uinteger_4 values – the TPer session number and the Host session number (Session = TPerSN concatenated with the HostSN). The TPer Session Number is sent first; the Host Session Number is second. Consequently, the same session number is used for communications between both parties.

- **SeqNumber: uinteger_4 –** An incrementing counter that starts at 1 and increments until 2^{32}-1, which identifies the number of the packet within the session and defines the ordering of transmitted packets.

  The message recipient shall ignore a packet with an equal or lower SeqNumber value than any previously acted-upon packet. In addition, wrapping of the SeqNumber shall result in the session being automatically aborted.

  Each communicator shall maintain multiple SeqNumber counts, including that of the last packet acknowledged, the next packet expected, and the last packet transmitted.
- **AckType: uinteger_2 –** This will be `0x00 0x01` if the Acknowledgement field is a packet acknowledgement. This will be `0x00 0x02` if the Acknowledgement field is a packet negative acknowledgement. This will be `0x00 0x00` if no packets are being acknowledged or negative acknowledged, and the value of the Acknowledgement field shall be zeroes.
- **Acknowledgement: uinteger_4 –** If the value of the AckType field is `0x00 0x01`, then this number shall be the SeqNumber of the last packet successfully received by the receiver. If the value of the AckType field is `0x00 0x02`, then this shall be the SeqNumber of the packet at which the receiver wishes the sender to begin retransmission. Generally, the receiver will put a value of the last known good packet received plus one. For AckType field value of `0x00 0x02`, the communicator shall not NAK a SeqNumber less than or equal to the last ACKed SeqNumber. If the AckType field is `0x00 0x00`, then the value of this field shall be zeroes.
- **Length: uinteger_4** – The number of bytes in the Payload field (made up of the IV field, Secure Data field and the Message Authentication Code field).

- **Payload**
  - **Initialization Vector (IV): uinteger{0-16}** – The IV input for the selected encryption or integrity checking mode. For GCM, GMAC, and CCM, the IV is 8 bytes long and shall contain a unique value with each encryption invocation. A simple algorithm is for the sender to use the sequence number as the IV. For AES-CBC encryption, the IV shall contain a random 16-byte integer. For all other modes, the IV shall have zero length.
  - **SecureData** – This is the encrypted/integrity-protected data. The Secure payload field s made up of the following parts (all of which are encrypted, if secure messaging requires encryption:
    - **DataLength: uinteger_4** – Length of the Data field, in bytes.
    - **Data:{Data Length}** – The encrypted or integrity-checked set of subpackets and any necessary padding (dependent on encryption mode). This field is made up of the following two parts:
      - **SubpacketData:{Data Length}** – The encrypted or integrity-checked subpackets.
      - **Pad: bytes**: - Any necessary padding required to fulfill the alignment constraints for the encryption mode in use. For AES-CBC encryption, the length of the Pad field shall include a number of padding bytes such that the total length of the Data field plus the Pad field is congruent to zero mod 16. For GCM and CCM, there is no required padding.
  - **Message Authentication Code (MAC): {size of MAC}** – A message authentication code that protects the integrity of the packet. The MAC covers the Session, SeqNumber, AckType, Acknowledgement, Length, IV, and, for encrypted data, the ciphertext (the value of the SecureData field, which is made up of the Data Length and Data fields; note the Data field is made up of the SubpacketData field and the Pad field), or, for unencrypted data, the unencrypted SecureData field.

3.2.3.6 Method Invocation – Result Retrieval Protocol

A method is invoked by tokenizing the method call and its parameters as described in previous sections, using the token encoding format and Subpacket-Packet-ComPacket format. The host sends the ComPacket to the TPer in an IF-SEND command. Multiple IF-SEND commands may be required to encompass the entirety of a method invocation or series of method invocations, and their related data.

The host then polls the TPer by transmitting IF-RECV commands. When the TPer has packaged its response, it transmits the tokenized results to the host in response to an IF-RECV command. Multiple IF-RECV commands may be required to retrieve all of the results of a particular method invocation or series of method invocations.

For additional information on the operation of the IF-RECV commands, see the descriptions for those commands as detailed in the appropriate interface specifications.

3.2.4 Templates

Templates are sets of tables and methods, grouped by feature, from which SPs are created.

This document covers the following Templates:

- **Base Template:** Provides the tables and methods common for all SPs.
- **Admin Template**: Provides administrative control over other SPs and the TPer settings as a whole, and control over Issuance of new SPs.
- **Clock Template:** Contains tables and methods specialized for forensic and cryptographic clocks.
- **Crypto Template:** Contains functional extensions to the Base SP cryptographic and procedural capabilities.
- **Locking Template:** Provides tables and methods for storage encryption/decryption and read/write lock state control.
- **Log Template:** Contains tables and methods specialized to forensic logging.

3.2.5 Tables - Details

All persistent data for SPs are stored in tables – the only data for an SP that persists past the end of a session is the data that is stored in tables. Tables survive operations on user-areas, such as reformatting.

Tables are stored in SP-specific parts of the secure storage area of the TPer. The secure storage area(s) of a TPer are only accessible by the T10 SECURITY PROTOCOL/T13 TRUSTED commands.

A table consists of a grid with named columns and addressable rows. At each column and row intersection there is a cell. All the cells in a column have the same type. The column types are specified at table creation. For some SSCs, the number of rows in a table is completely determined when it is created (additional rows cannot be allocated), but other SSCs define tables with a dynamically allocable number of rows. If an SSC permits additional rows to be added to a table, then the number of rows specified at table creation is the initial number of rows allocated for that table.

A table name or table column name may be up to 32 bytes in length. By convention, the names assigned in this document consist of ASCII characters, the first of which is a letter and others is a letter, digit or underscore. Adjacent underscores do not occur. All names are case sensitive.

SPs may be issued and deleted. Within an SP, tables may be created and deleted. For each table, rows may be created and deleted (except within a Byte table – see 3.2.5.1), but columns are created only when the table is created. A specific Security Subsystem Class may disallow the creation of any of these.

Each SP has a set of metadata tables (such as the Table table, Column table, etc.) that describes all the tables of the SP including the metadata tables themselves.

Access control provides a means to limit the methods that may be executed on tables, or particular rows or cells of tables.

3.2.5.1 Kinds of Tables

There are three kinds of tables:

1. **Byte table.** A byte table has one unnamed column of type uinteger_1. Note: The rows of this table cannot be allocated or freed. The address of the first row in a byte table is 1. Byte tables provide raw data storage.
2. **Array table.** Unlike byte tables, array tables may have more than one column. Each array table row is addressed by an unsigned integer that is stored in a column, named RowNumber, which is only readable from the host – the host shall not specify or modify the value of this column for a row in an array table. The first row in an array table has RowNumber column value 1, etc. Array tables provide storage for categorizable data.
3. **Object table.** When created, one or more columns are designated as the index. Each row of the table has a value or combination of values in the indexed column(s) that is unique within the table for those column values. When more than one column is marked for indexing, each indexed column participates in the index ("multi-column index"). The TPer is not required to keep rows of the table sorted by the index. Object tables provide storage for data that binds a set of methods to that data.

For Object and Array tables:

- A newly created table is initially empty and rows must be created before they can be used.
- There is always a UID column of type UID. In object tables, rows are addressed by UID. In array tables, rows are addressed by row number or UID.

3.2.5.2 Objects

An object is any row of an object table. The particular object type is defined by the object table in which the object occurs. The columns of the object table define the contents of each object in it.

For example, object table Foo is the Foo object type. Each row in object table Foo is an instance of the Foo object type.

An important aspect of an object type is the set of methods it defines. For a specific SP, there are methods on the SP itself, methods that act on the tables and have the whole table as their possible scope, and methods for each of the objects within the SP. In order to hide the information inside an object, object-specific ACLs are applied to the methods capable of manipulating that object's data.

3.2.5.3 Unique Identifiers (UIDs)

Each array and object table has a column named UID. This column contains an 8-byte unique identifier for that row. Each row has an SP-wide unique value in this column. This value is never shared with another row, and is never reused by that SP. The TPer shall guarantee that UIDs are unique across the entire SP anytime that a UID is generated.

The UID column is present to provide anti-spoofing capability, and to provide a means to address these rows. New UIDs are assigned when rows are created and old values are discarded when rows are deleted. If all UIDs have been used, no more rows can be created.

Each table is also represented by a UID. A table's UID is derived from the UID of that table in the Table table. The Table table is an object table in which each row is a table descriptor object that stores metadata about the associated table.

The bytes in a UID shall be utilized as follows:

- The first four bytes of a table row's UID shall be the "containing table" id and the last four bytes will be assigned in a TPer-specific manner.
- UIDs of tables shall be assigned as follows:
  - The UIDs of table descriptor objects (the table's row in the `Table` table) shall be `0x00 0x00 0x00 0x01 XX XX XX XX`, where `XX XX XX XX` represents the values assigned by the TPer to that object's UID. For example, The `Table` table's UID shall be `0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x01`
  - The UID used to reference the actual table (rather than that table's row in the `Table` table) shall be `XX XX XX XX 0x00 0x00 0x00 0x00`, where `XX XX XX XX` are the last four bytes of that table's UID in the `Table` table. Four `0x00`'s as the last four bytes of a UID that does not have four `0x00`'s at the beginning are references to a table. This is the UID that is returned by a successful invocation of the `CreateTable` method.
  - All non-table UIDs shall have their high four bytes be the low four bytes of the containing table's UID. So, references to rows in a table are assigned UIDs based on the UID of the containing table. For instance, references to the rows in table `XX XX XX XX 0x00 0x00 0x00 0x00` are assigned UIDs `XX XX XX XX yy yy yy yy` where the first four bytes of the containing table UID and of the row are the same.

All UIDs with their first four bytes equal to `0x00 0x00 0x00 0x00` are reserved for use by the TCG and shall never be assigned by the TPer. When necessary to refer to the SP with a UID, a UID of `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01` is reserved to signify "this SP".

For each table defined in this specification, UIDs with last four bytes between `0x00 0x00 0x00 0x00` and `0x00 0x01 0x00 0x00` shall be reserved for use by the TCG.

A Null UID reference is all zeroes (`0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00`).

3.2.6 Common Methods

Each table and each object has these methods:

- **Get:** Used to access the values of one or more table cells.
- **Set:** Used to modify the values of one or more table cells.

Object and array tables also have these methods

- **CreateRow:** Used to insert a new row into the table.
- **DeleteRow:** Used to delete one or more rows of a table.
- **Next:** Used to iterate over all the rows of a table.

The `CreateRow`, `DeleteRow`, and `Next` methods are not defined for byte tables or for objects.

3.2.7 SP Tables & Method Summary

The SPs, Tables, and Methods are presented in overview in Table 22 below.

Table 22 SPs and Methods Covered in this Document

Template	*Grouping Concepts*	Tables	Table Type	*Comments*		*Grouping Concepts*	Methods	*Comments*
Base	*All SPs Have Base*							
	Version, Size, Etc.	**SPInfo**	Array	*Details of SP*		*Basic SP*	**DeleteSP**	*SP Deletion*
		SPTemplates	Array	*SP Components*				

	Tables and Methods	**Table**	Object	*Tables in SP*	*Basic Table/Object*	**CreateTable**	*Table Creation*
		Column	Array	*Cols in Tables*		**Delete**	*Object Deletion*
		Type	Object	*SP Types*		**CreateRow**	*Row Creation*
		MethodID	Array	*Methods in SP*		**DeleteRow**	*Row Deletion*
		Method	Array	*Access Control Associations*		**Get**	*Read Cells*
	Access Control	**ACE**	Object	*Access Control on Methods*		**Set**	*Write Cells*
		Authority	Object	*Authorities in SP*		**Next**	*Next Row*
		Certificates	Object	*Certs for Public Key Credentials*	*Basic Method*	**DeleteMethod**	*Method Deletion*
						Authenticate	*Authenticate Authority*
	Credentials	**C_PIN**	Object	*Credential*	*Access Control*	**GetACL**	*Set ACEs on Method*
		C_RSA_1024	Object	*Credential*		**AddACE**	*Create ACEs*
		C_RSA_2048	Object	*Credential*		**RemoveACE**	*Delete ACEs*
		C_AES_128	Object	*Credential*	*Misc*	**GenKey**	*Generate Keys*
		C_AES_256	Object	*Credential*			
		C_EC_160	Object	*Credential*			
		C_EC_192	Object	*Credential*			
		C_EC_224	Object	*Credential*			
		C_EC_256	Object	*Credential*			
		C_EC_384	Object	*Credential*			
		C_EC_521	Object	*Credential*			
		C_EC_163	Object	*Credential*			
		C_EC_233	Object	*Credential*			
		C_EC_283	Object	*Credential*			
		C_HMAC_160	Object	*Credential*			
		C_HMAC_256	Object	*Credential*			
		C_HMAC_384	Object	*Credential*			
		C_HMAC_512	Object	*Credential*			
Template	*Grouping Concepts*	**Tables**	Table Type	*Comments*	*Grouping Concepts*	**Methods**	*Comments*
Admin				*Stores TPer/SP Info*			
	Basic Info about TPer	**CryptoSuite**	Array	*Crypto Capability*	*Issuance*	**IssueSP**	*Issue SP*
		TPerInfo	Array	*Details of TPer*			
		Properties	Byte	*TPer communications details*			
	SPs on TPer	**Template**	Object	*SP Templates*			

		SP	Object	*Issued SPs*			
Template	*Grouping Concepts*	**Tables**	Table Type	*Comments*	*Grouping Concepts*	**Methods**	*Comments*
Clock				*Keeps date/time*			
	Clock Information	**ClockTime**	Array	*Holds all clock info*	*Clock Management*	**GetClock**	*Reading Clock*
						ResetClock	*Managing Clock*
						SetClockHigh	*Sets time from high trust source*
						SetLagHigh	*Sets lag time from high trust source*
						SetClockLow	*Sets time from low trust source*
						SetLagLow	*Sets lag time from low trust source*
						IncrementCounter	*Reading Monotic Counter*
Template	*Grouping Concepts*	**Tables**	Table Type	*Comments*	*Grouping Concepts*	**Methods**	*Comments*
Crypto				*Enable Hidden CSP*			
	Hash Functionality	**H_SHA_1**	Object	*Credential*	*Crypto Operations*	**Random**	*Gen Random Number*
		H_SHA_256	Object	*Credential*		**DecryptInit**	*Public or Symmetric Key Decryption*
		H_SHA_384	Object	*Credential*		**Decrypt**	*Public or Symmetric Key Decryption*
		H_SHA_512	Object	*Credential*		**DecryptFinalize**	*Public or Symmetric Key Decryption*
						EncryptInit	*Public or Symmetric Key Encryption*
						Encrypt	*Public or Symmetric Key Encryption*

						EncryptFinalize	*Public or Symmetric Key Encryption*
						HashInit	*Hash*
						HashCalc	*Hash*
						HashFinalize	*Hash*
						HMACInit	*HMAC*
						HMACCalc	*HMAC*
						HMACFinalize	*HMAC*
						Sign	*Public Key Sign*
						Verify	*Public Key Verify*
						XOR	*For Key Derivation*
Template	*Grouping Concepts*	**Tables**	Table Type	*Comments*	*Grouping Concepts*	**Methods**	*Comments*
Log	*Forensic Logging*						
	Logging	**Log**	Array	*Stores logs*	*Log Management*	**AddLog**	*Add record to Log Table*
		LogList	Object	*Contains Log table metadata*		**CreateLog**	*Create a new Log Table*
						ClearLog	*Removes all entries in a Log Table*
						FlushLog	*Commits log entries in main memory*
Template	*Grouping Concepts*	**Tables**	Table Type	*Comments*	*Grouping Concepts*	**Methods**	*Comments*
Locking	*Encryption/Key Management/Read-Write Lock State Control*						
	Device Management	**LockingInfo**	Array	*Device capability*	*Key Management*	**GetPackage**	*Wrapped Key Retrieval*
		Locking	Object	*LBA Ranges definitions*		**SetPackage**	*Wrapped Key Retrieval*
	Boot Control	**MBRControl**	Array	*Boot Control*			
		MBR	Byte	*Boot Control Code*			
Template					*Grouping Concepts*	**Methods**	*Comments*
None							

	Session Manager Layer Methods *- These methods are not associated with a particular Template or SP. These methods provide the host the capabilities required to start sessions with SPs.*		*Session Management*	**Properties**	*Channel information*
				StartSession	*Session Startup*
				SyncSession	*Session Startup*
				StartTrustedSession	*Secure Session Startup*
				SyncTrustedSession	*Secure Session Startup*
				CloseSession	*Session Termination by the TPer*

3.3 Interface Communications

The TCG Core Specification describes the architecture and main command set in an interface protocol-independent way. The Core Specification is not, however, agnostic to the interface protocols. It recognizes the limitations and characteristics of the main targeted interface protocols. In particular the TCG has targeted the INCITS T13/ATA and INCITS T10/SCSI protocols. Accordingly, the TCG has secured a set of command codes from these standards bodies that will allow the current specification to be implemented. These commands are the T10 SECURITY PROTOCOL IN/OUT and the T13 TRUSTED SEND/RECEIVE.

This section abstracts out the common features of these commands that will serve as a requirement for an interface protocol to implement the present specification.

The following assumptions are made regarding the interface commands:

- The interface commands have two parts: (1) a command block and (2) a data block or payload. The data blocks for a particular interface protocol are of a fixed size, called BLK-SIZE. BLK-SIZE must be at least 512 bytes. The data block size for a particular interface protocol is assumed to be fixed and therefore the parameter BLK-SIZE is not part of the command block.
- There is at least one command in the interface protocol that transfers data from the host to the storage device. These commands are called IF-SEND.
- There is at least one reserved command in the interface protocol that transfers data from the storage device to the host. These commands are called IF-RECV.
- The command blocks for these two commands shall have the following fields
    - Protocol ID: with at least 6 values that can be mapped into 1 to 6.
    - Transfer Length: 2 bytes, indicating the number of blocks to be transferred.
    - ComID: 2 bytes, the ComID to be used.
- The interface protocol shall preserve the order for IF-SEND and IF-RECV. That is, the commands sent from a particular host will arrive at the TPer in the order in which they were sent from that host.

The command block of the interface commands are described in the format defined in Table 23.

Table 23 Interface Command – Command Block

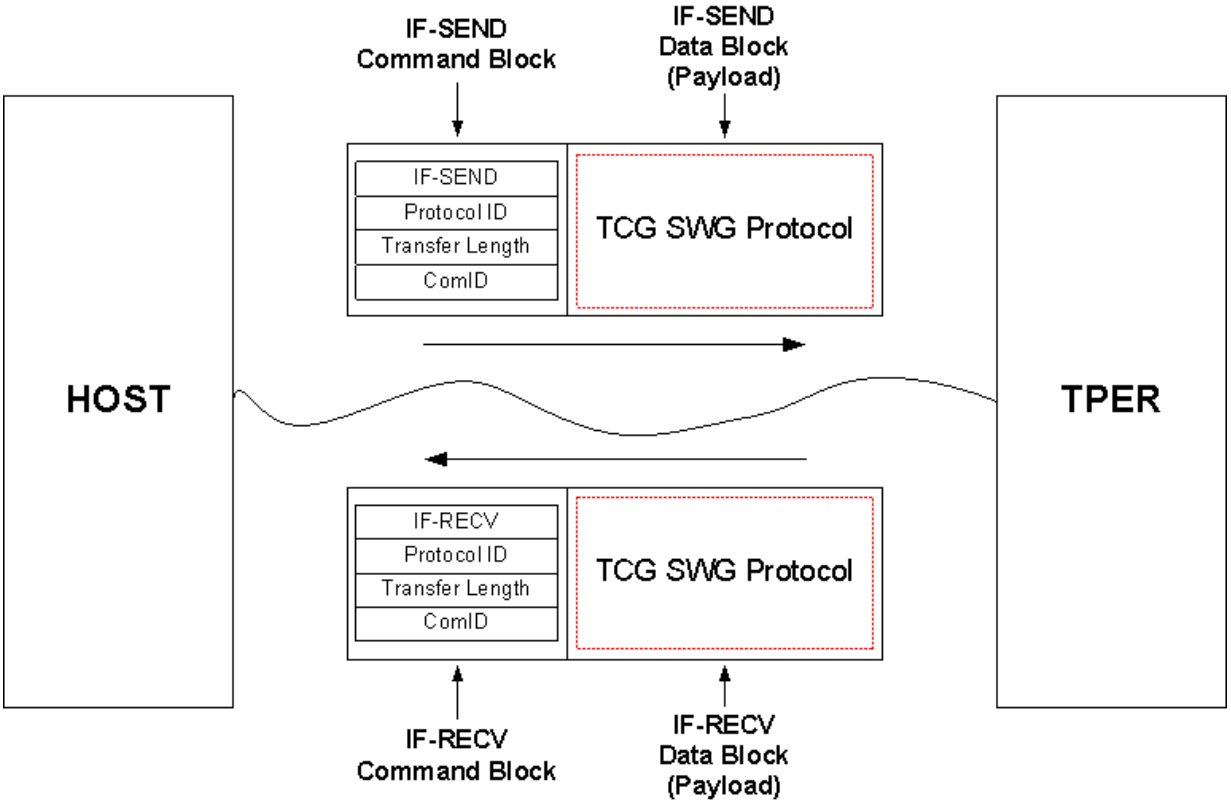
Command	Either IF-SEND or IF-RECV.
Protocol ID	Between 1 and 6
Transfer Length: 2 bytes	The number of blocks to be transferred.
ComID: 2 bytes	The ComID to be used.

The mapping of the IF-SEND and IF-RECV commands to specific interface protocol commands are described in the TCG documents related to the particular protocol.

3.3.1 Communicating With the TPer Through the Interface Protocol

The communication between the Host and the TPer take place through the use of IF-SEND and IF-RECV as illustrated in Figure 4 below.

Figure 4 TPer-Host Communication



Most of the useful communication is encapsulated in the payload of these commands. However, there will be some interaction with the command block as well, particularly for the lower level commands used to start a session.

The payload of these commands shall be of one of two types

- Packetized payloads: The data block has a single ComPacket. The ComPacket contains one or more packets. Each packet contains one or more subpackets. Details of how the packets are specified in section 3.2.3.4.
- Byte field payloads: This is a simple type of payload that is used in the more rudimentary layers of the protocol stack defined in this section.

The packetized payloads are used when the Protocol ID of the command block is set to 1, and the byte field payloads are used when the Protocol ID is set to 2. The transfer length varies depending on the commands.

The ComID are used to allow the TPer to identify the caller of the IF-RECV command and appropriately populate the payload for the command.

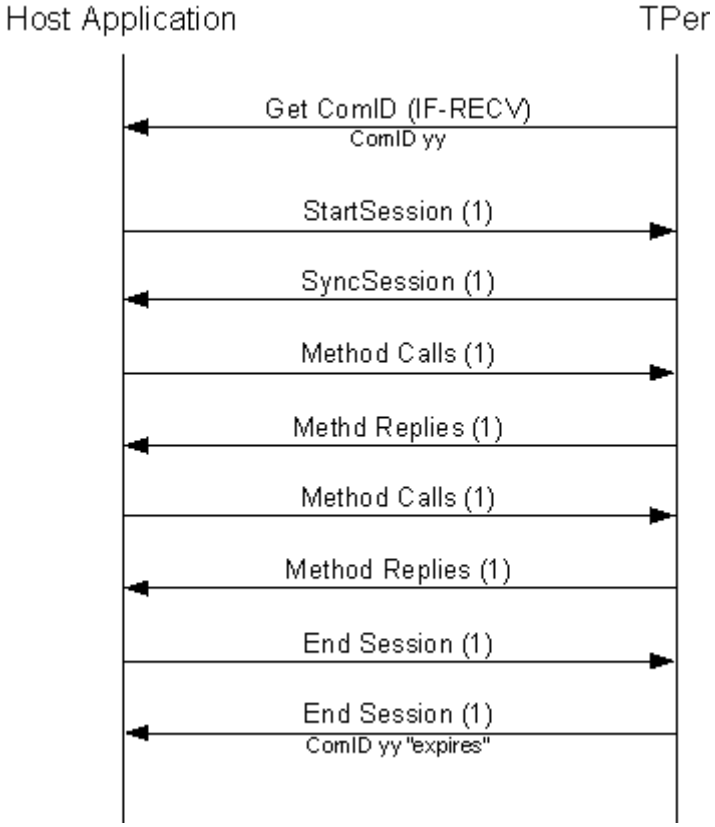
3.3.2 The ComID

The ComID is used to enable the correct communication of response data to the host. The ComID allows the TPer to identify the caller of the IF-RECV command and appropriately populate the payload for the command.

In order to open a session with a particular SP on a TPer, the host application must start by requesting a ComID from the TPer. The TPer issues a ComID to the host application. The ComID is transmitted in the Security Protocol Specific field of the interface command. Once the host application has a unique

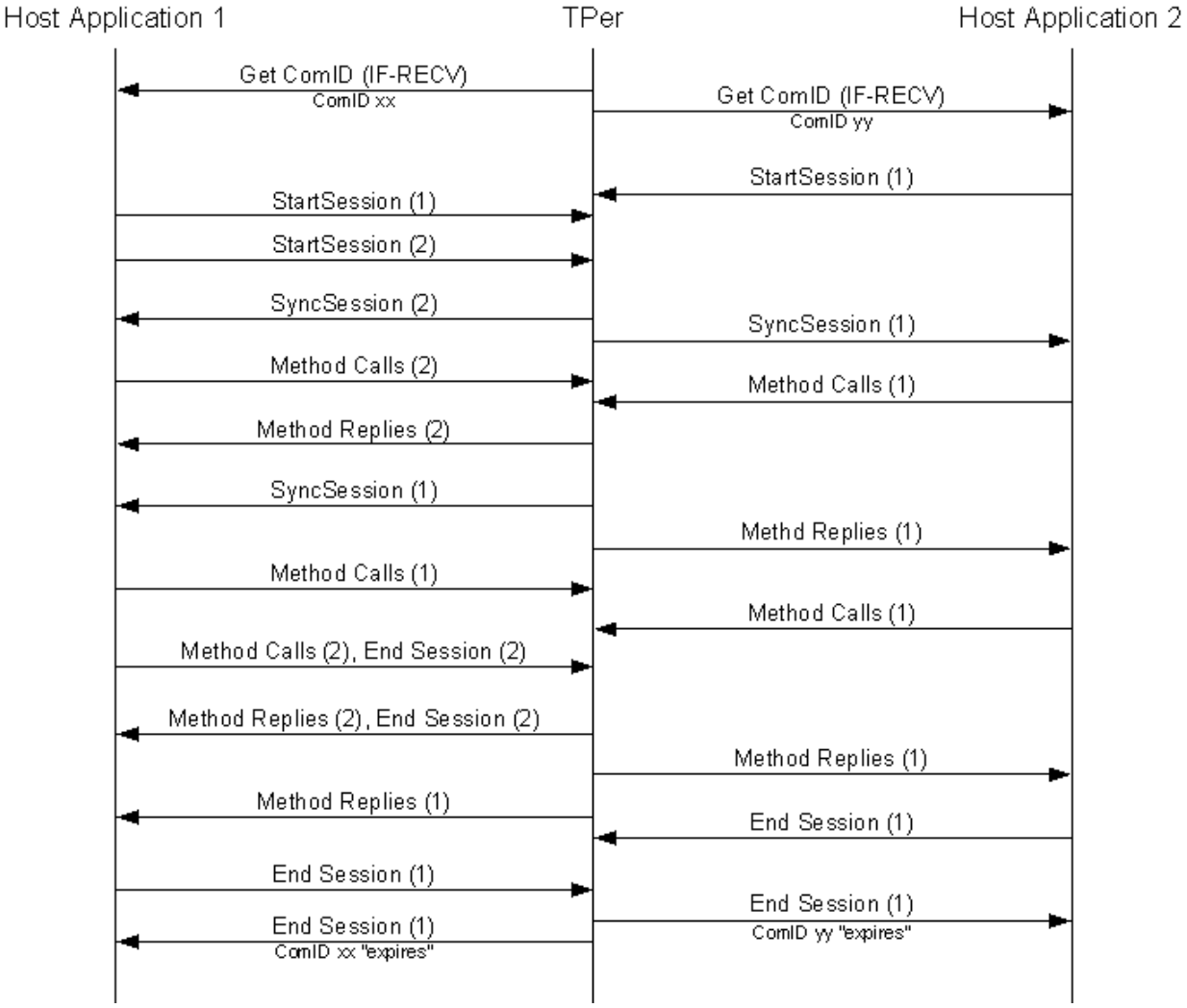
ComID, the host is able to initiate the process of starting a session. An example of this interaction can be found in Figure 5 .

Figure 5 Single Host/TPer Interaction



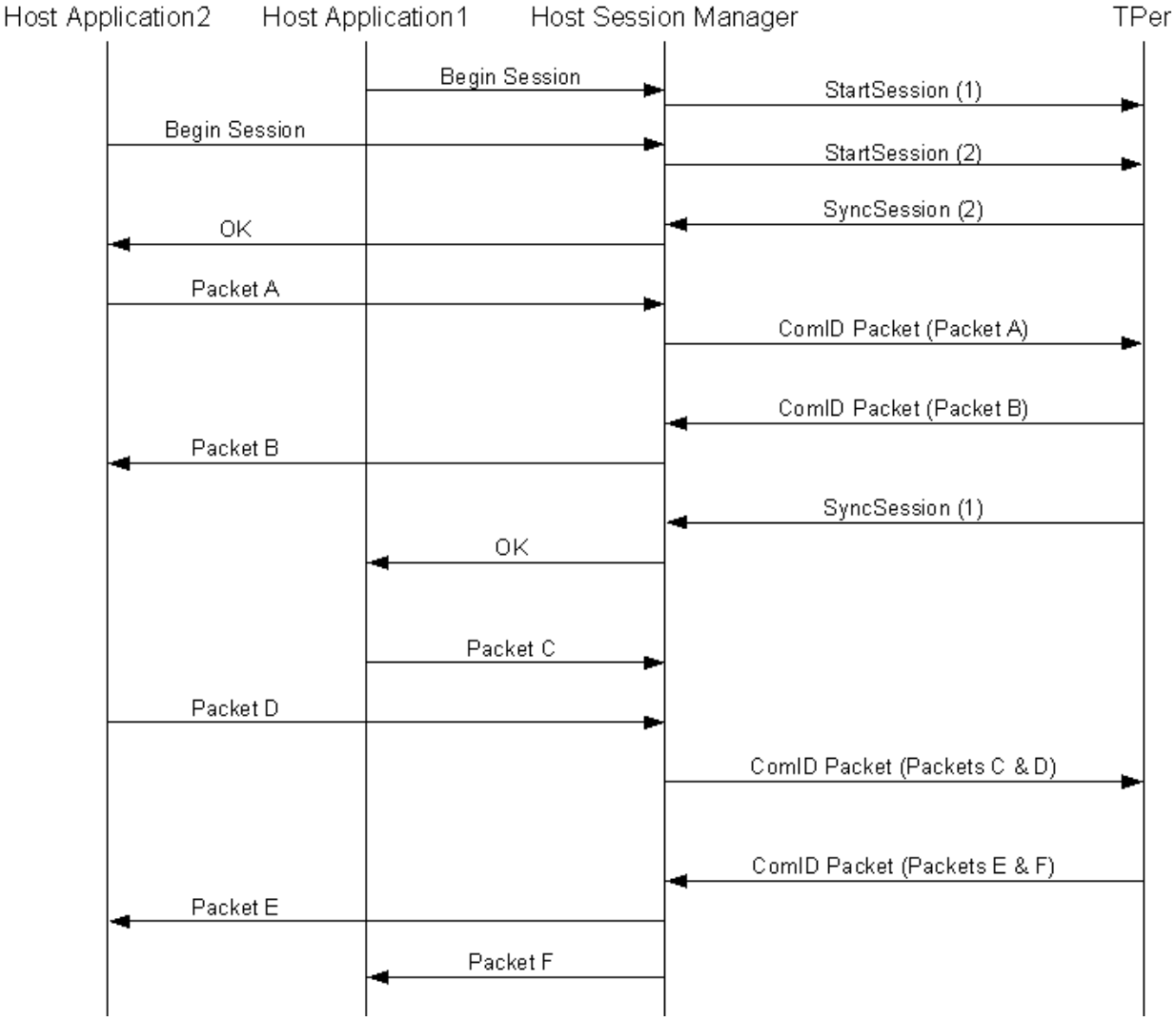
Once the session is started, the TPer associates the session number with the ComID. In this way, when an IF-RECV is sent to the TPer using Protocol ID of 1, the TPer is able to respond with a payload containing only the packets for the session number associated with the ComID. This allows for multiple applications to be simultaneously communicating with the TPer without interfering with one another, as displayed in Figure 6 .

Figure 6 Multiple Host/TPer Interaction



In some situations it might be useful to allow for a single entity, called the Host Session Manager, to manage the TPer communications for a set of different applications running on the host. To enable this, multiple sessions are permitted to be opened with a single ComID. All the sessions opened with a given ComID shall be associated with it. An example of this behavior is displayed in Figure 7 .

Figure 7 Host Session Manager/TPer Interaction



Note that to the TPer, communication with a single application is no different than communication with a Session Manager that may act as an intermediary for multiple applications with which the TPer is communicating. An application may open a single session to the TPer for itself, multiple sessions for itself, multiple sessions for one or more other applications, multiple sessions for itself and one or more other applications, or any other combination.

When an IF-RECV is sent to the TPer using a particular ComID, the TPer shall respond by putting packets from the sessions associated with the ComID into the response. If there are more pending responses from the various sessions associated with the ComID than fits the IF-RECV, it is up to the TPer to determine which packets to include.

The number of packets/subpackets that are included in the response is a function of the amount of available responses, the transfer length of the command, and the flow control mechanism. The amount of data still remaining to be retrieved and the minimum transfer length required to retrieve at least one packet, at the time the ComPacket was generated, is reported in the ComPacket header.

3.3.3 ComID Management

A mechanism is required to manage the ComIDs so as to minimize the chances of two host applications using the same ComID in the rare occasions in which there are ComID conflicts.

ComIDs may be of two types: reserved and normal. The reserved ComIDs are used to allow for rudimentary commands to be defined at lower levels of the protocol stack. One example is the GET_COMID command (see definition below). The lower 4096 out of the possible ComIDs shall be reserved – 0-2047 are reserved for TCG use/assignment, and 2048-4095 are reserved as vendor-unique. The other, non-reserved ComIDs shall be used for multiplexing the TPer responses to IF-RECVs.

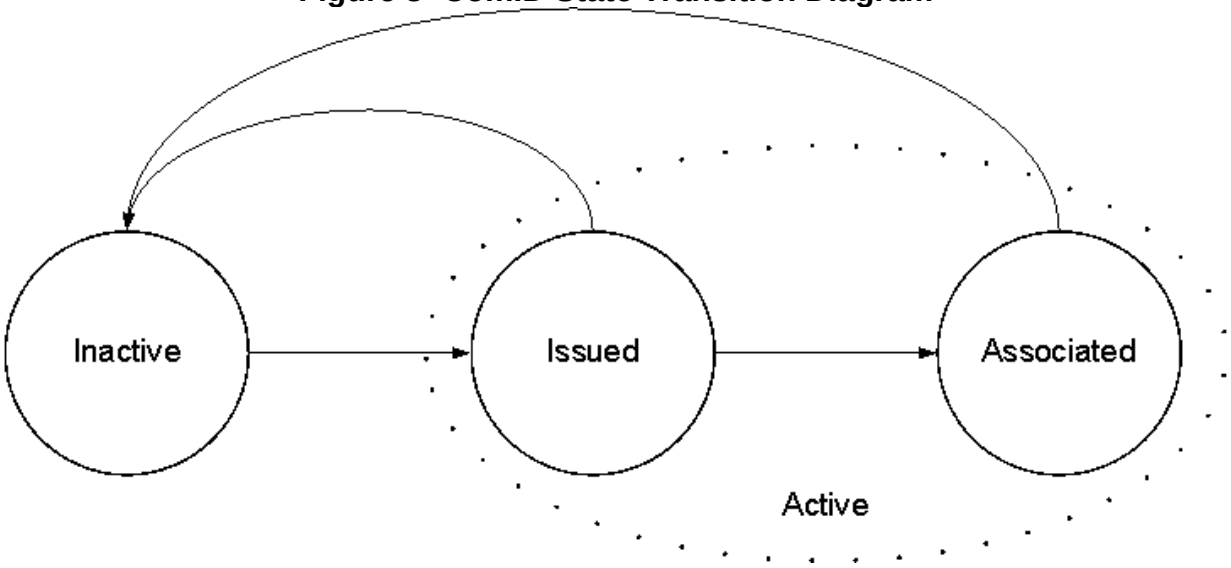
A ComID may be in one of the following three states:

1. **Inactive**: The ComID has not been assigned to anyone since the last hardware reset or power cycle; ComIDs may also be in the Inactive state due to circumstances other than reset/power cycle.
2. **Issued**: The ComID has been issued (it was returned to the host during a successful completion of a GET_COMID command) but no sessions have been started using this ComID.
3. **Associated**: One or more open sessions are associated with the ComID.

ComIDs that are either in the Issued state or the Associated state are said to be Active. The state diagram in Figure 8 shows these states and the possible transitions among them.

Note that support for ComID management commands is SSC-dependent.

Figure 8 ComID State Transition Diagram



The possible state transitions are:

- **Inactive to Issued**: A ComID transitions from the Inactive state to the Issued state when it is returned to the host during a successful execution of the GET_COMID command.
- **Issued to Associated**: A ComID transitions from the Issued state to the Associated state once a session is open using that ComID. This occurs when at the point when session startup has successfully completed.
- **Issued to Inactive**: A ComID transitions from Issued to Inactive when any one of the following conditions hold
  - There is a hardware reset or power cycle.
  - The host issues MAX_COMID_CMD commands using the ComID before starting a session. MAX_COMID_CMD defines a limit on the number of commands that a TPer will accept for a given ComID before the TPer transitions an Issued ComID to Inactive.

A TPer's MAX_COMID_CMD value is retrieved using the `Properties` method. Support for MAX_COMID_CMD is SSC-dependent.

- The host does not start a session using the ComID within MAX_COMID_TIME from the ComID being issued. MAX_COMID_TIME defines a limit on the amount of time a ComID can exist in the Issued state without an active session. A TPer's MAX_COMID_TIME value is retrieved using the `Properties` method.

- **Associated to Inactive**: A ComID transitions from Associated to Inactive when any of the following conditions are met:
  - There is a hardware reset or power cycle.
  - After all sessions associated with the ComID are closed.

In order to minimize the possibility of conflict, the ComID issuance mechanism shall have the following two characteristics:

- A ComID that is in an active state shall not be issued again. That is, only ComIDs that are in either the inactive or issued states shall be returned to the host as a response to the Get_COMID command.
- The TPer shall issue ComIDs in a sequential manner (wrapping around cyclically as needed), keeping a record in non-volatile memory that is a pointer to a ComID not yet allocated.

In addition to the above transitions, the TPer may transition a ComID to the Inactive state at any time for any reason.

3.3.3.1 Extended ComID

Despite all the mechanisms put in place, there is always the possibility that some application will hold on to its ComID for an extended period of time and not recognize that the ComID has become Inactive and (possibly) subsequently been issued to another application. Since there are only 61440 normal non-reserved ComIDs, the probability of this occurring is not small enough to be neglected.

To help deal with this issue the TPer makes use of Extended ComIDs. Extended ComIDs are 4 bytes long and have the first 2 bytes equal to the ComID. The MSB of the ComID is the first byte of the Extended ComID, and the LSB of the ComID is the second byte of the Extended ComID. The TPer arbitrarily generates the remaining 2 bytes every time a ComID is issued. The GET_COMID command will return the 4-byte Extended ComID to the host. Note that there may have been many Extended ComIDs associated with the same ComID over the life of the TPer. Extended ComIDs associated with reserved ComIDs (0-4095) shall always be 0.

The Extended ComID can be in one of the following states

1. **Inactive**: The associated ComID is in the inactive state.
2. **Issued**: The Extended ComID has been issued (it was returned to the host during a successful completion of a GET_COMID command) but no sessions have been started using the associated ComID.
3. **Associated**: One or more open sessions were open with the ComID. These sessions are said to be associated with the Extended ComID.
4. **Invalid**: The Extended ComID has not been issued since the last power cycle/reset, or has become inactive and there exists another Extended ComID with the same associated ComID in one of the active states (Issued or Associated).

The Extended ComID can be used to determine if an application is using a stale, conflicting ComID, i.e., if the ComID the application is using has become inactive and subsequently assigned to another application. When this happens, the application's Extended ComID shall be invalid. When the application makes an inquiry to the TPer using the Extended ComID, the TPer shall respond with an indication that the Extended ComID is invalid.

3.3.4 Sessions

There are currently two types of sessions:

- Regular Sessions (or just Sessions): These are communication channels between a host application and an SP.
- Control Sessions: These are between the TPer Session Manager (TSM) and the Host Session Manager (HSM).

The Host Session Manager is an abstract entity that represents the peer, on the host side, of the TPer Session Manager. The HSM could be an application that is routing traffic to several applications on the host or it could simply be a module in a given application that deals with establishing sessions with the TPer.

3.3.4.1 Regular Sessions

Each Regular Session is identified by a distinct Session Number (SN). The SN is an 8-byte quantity composed of two subparts: the TPer Session Number (TSN) and the Host Session Number (HSN), each of which has 4 bytes.

SN = (TSN, HSN)

The HSN is assigned by the HSM and can be any value. Typically the HSM will assign HSNs in such a way as to make them unique for all of its communications with one or more TPers, though this is not required.

The TSN is assigned by the TSM. The TSM shall guarantee that all Regular Sessions associated with a particular ComID are assigned a different TSN. In addition, the TSM shall not assign any TSN in the range 0 to 4097 to a regular session. These TSNs are reserved by TCG for special sessions, of which the control session is the only one currently defined.

Additional details regarding session startup can be found in 3.4.4.5.

3.3.4.2 Control Sessions

Each Control Session is identified by a distinct ComID and a TPer Session Number of zeroes (TSN = `0x00 0x00 0x00 0x00`). There is exactly one Control Session associated with each ComID. The Control Session is between the TSM, identified by TSN=`0x00 0x00 0x00 0x00` and the HSM, identified by the ComID. The HSN is not used to identify the session in a Control Session. It is used simply as a "routing aid" for the cases in which multiple sessions are simultaneously started using the same ComID. However, the HSN will become active once the Regular Session is started.

The life cycle of the Control Session is tied to the life cycle of the ComID in that the Control Session associated with a particular ComID starts as soon as the ComID is issued and it starts with the default credits. When the ComID is retired, the Control Session is terminated. The flow control for the Control Session is performed in the same manner as the flow control for Regular Sessions, with the difference that the communication is between the TSM and the HSM and these entities are responsible for the flow control.

The packet headers for communications in the case where several different sessions are simultaneously started on the same ComID will have different SNs because the HSNs will be different. As far as flow control is concerned, only the TSN matters since the Control Sessions are identified by the TSN=`0x00 0x00 0x00 0x00` and the ComID.

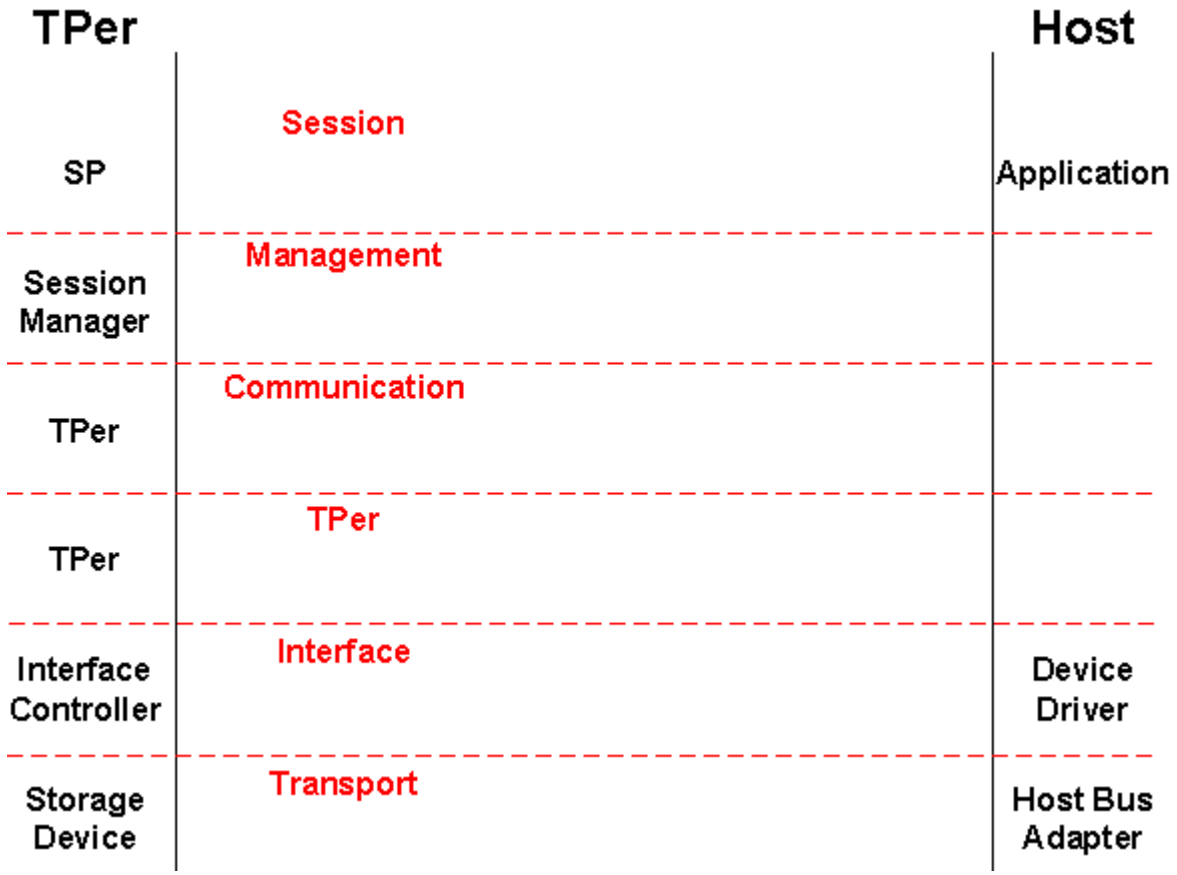
3.3.5 Protocol Layers

In order to describe the overall process for establishing communication with the TPer and initiating a session to an SP, it is necessary to partition the protocol stack into layers. The commands in each

layer differ in the amount of functionality available. The lower level allows only one-way communication (TPer to host) and shall only use simplistic byte field responses. The higher layers have two-way communication and use packets and methods.

Figure 9 below depicts the protocol layers.

Figure 9 TPer-Host Communication Protocol Layers



- **Session layer**: This layer is entered when a session is successfully established between the host application and an SP in the TPer. Most of the commands and functionality specified in the TCG Core Specification operate in this layer. Payloads in this layer are packetized and tokenized.
- **Management layer**: This layer deals with establishing a session between an SP and a host application. Payloads in this layer are packetized and tokenized.
- **Communication (Com) layer**: In this layer the host application already has an assigned ComID that is used for establishing two-way communication. It is a bidirectional communication/control layer. This layer is used for management of ComIDs and dealing with error conditions and other storage device management issues.
- **TPer layer**: This is the first entry point to the TPer. This is a "one-way" communication layer. That is, only IF-RECV commands are dealt with in this layer. The host application does not have a ComID yet. There is a set of reserved ComIDs that can be used to execute special commands at this layer.

- **Interface layer**: This portion of the stack contains the protocol for allowing the host to control a specific storage device. The interface protocol must support IF-SEND and IF-RECV, i.e., have commands with the properties that are required for these TCG commands.
- **Transport layer**: This portion of the stack is responsible for transporting the data from one particular host to one particular storage device and vice-versa. An example is Fibre Channel.

3.3.5.1 Transport Layer

This layer of the protocol stack is responsible for transmitting the data from one particular host to one particular storage device and vice-versa. There are no specific interactions with this layer described in the TCG Core Specification. The only requirement is that this layer interact with the Interface layer in such a way as to guarantee that the order of commands sent from a single host to a single storage device are preserved.

3.3.5.2 Interface Layer

The commands at this layer are the IF-SEND and the IF-RECV commands. The interface controller on the storage device shall identify these commands and send them to the TPer level.

All commands that map to IF-SEND and all the commands that map to IF-RECV that have the protocol ID field in the set {1, 2, 3, 4, 5, 6} shall be sent to the TPer.

For an IF-SEND, the interface controller retrieves the data from the host to the storage device, sends the command block parameters (Command, Protocol ID, Transfer Length, ComID) to the TPer, waits for all data to be transferred, and then returns the IF-SEND command status at the interface protocol level.

For an IF-RECV, the interface controller sends the command block parameters (Command, Protocol ID, Transfer Length, ComID) to the TPer, waits for data to be generated from the TPer, and starts transfer to the host. Once all the data has been transferred, the IF-RECV command status is returned at the interface protocol level.

3.3.5.3 TPer Layer

This is the entry point into the TPer. This layer has very limited functionality. Commands at this layer are designed to be used without ComIDs. In particular, the command used to request a ComID, GET_COMID, is dealt with in this layer.

The only commands dealt with in this layer are IF-RECV commands with some specific reserved ComIDs and protocol ID settings. All other commands are passed up to the Communication Layer. The commands specified in this layer and in the communication layer will have Protocol ID = 02h.

3.3.5.3.1 GET_COMID

The command block for the GET_COMID command is defined in Table 24. The payload of the GET_COMID command is defined in Table 25.

Table 24 GET_COMID Command Block

FIELD	VALUE
Command	IF-RECV
Protocol ID	02
Transfer Length	00 01
ComID	00 00

Table 25 GET_COMID Payload

BYTE	FIELD	VALUE

BYTE	FIELD	VALUE
0 – 3	Extended ComID	Allocated ComID
4 – (BLK-SIZE-1)	Reserved	zero

- The first 4 bytes of the payload shall be the Extended ComID. The first two bytes of the Extended ComID are the ComID. If the TPer is not able to assign a new ComID for any reason it will return all zeroes in the Extended ComID field.
- The remaining BLK-SIZE - 4 bytes of the single transferred data block shall be reserved and set to zero.

3.3.5.4 Communication Layer

The Communcation Layer provides a mechanism for two-way communication between the host application and the TPer. The primary purpose of the communication at this layer is to manage the allocated ComID and to verify the validity of the allocated ComID.

Communication at this layer occurs using IF-SEND and IF-RECV commands using Trusted Protocol ID 02h. The host must have a ComID that has been assigned by the TPer using the GET_COMID command available at the TPer Layer.

If the host application uses a ComID that is not valid or has become invalid since its last usage, the protocol at this layer has the ability to signal the error to the host application without raising exceptions on lower layers such as the Interface or TPer layers. This allows host applications to verify validity of ComIDs without disturbing the operation of the TPer.

3.3.5.5 Communication Layer Protocol

The commands for communication with the TPer at this layer are as follows:

- HANDLE_COMID_REQUEST: IF-SEND to ComIDs with the caller's Extended ComID passed as the first 4 bytes of the payload.
- GET_COMID_RESPONSE: IF-RECVs on ComIDs previously allocated by the TPer.

For any given ComID, the host is expected to issue request and response commands in pairs. Consecutive response commands return data corresponding to the last request received from the TPer. The response may be regenerated by the TPer at the time of receipt of the command.

3.3.5.5.1 HANDLE_COMID_REQUEST

This command is used to inquire about or manage the state of the ComID previously allocated by the TPer. The command block for the HANDLE_COMID_REQUEST command is defined in Table 26.

Table 26 HANDLE_COMID_REQUEST Command Block

FIELD	VALUE
Command	IF-SEND
Protocol ID	02
Transfer Length	nn nn
ComID	Allocated ComID

The payload sent by the host to the TPer, at the minimum, consists of the 4-byte Extended ComID and a Request code. Additional fields may be required for some request codes. Currently only one Request code is defined: Verify ComID Valid. The payload required for this request is defined in Table 27.

Table 27 Verify ComID Payload

BYTES	FIELD	VALUE
0 – 3	Extended ComID	Allocated ComID
4 – 7	Request Code	00 00 00 01
8 – n*BLK-Size	Reserved	zero

On receiving this request, the TPer checks if the ComID sent in the payload matches any of the ComIDs currently active in the TPer. The response is reported in the payload of the next GET_COMID_RESPONSE command sent to the requested ComID.

3.3.5.5.2 GET_COMID_RESPONSE

This command is used to pick up the response of the TPer to a previous HANDLE_COMID_REQUEST command. The command is sent to the ComID for which the status is requested. The command block for the GET_COMID_RESPONSE command is defined in Table 28

Table 28 GET_COMID_RESPONSE Command Block

FIELD	VALUE
Command	IF-RECV
Protocol ID	02
Transfer Length	nn nn
ComID	Request_ComID

The transfer length is the maximum length in blocks that the TPer may send in response to the command. If the actual length of the response data is smaller, then the TPer shall pad the data with zeros. If the actual length of the response data is larger, then the TPer shall only send the requested amount of data.

The payload of the response data always contains the maximum available length of the response data in byte 8-9. The host may use this information to repeat the response command with a transfer length that fits the available data.

Currently, only the response to the Verify_ComID_Valid request code is defined. The payload built by the TPer in response to the Verify_ComID_Valid command is defined in Table 29.

Table 29 Verify_ComID_Valid Command Response

BYTE	FIELD	VALUE
0 – 3	Extended ComID	Allocated ComID
4 – 7	Request Code	00 00 00 01
8 – 9	Available Data Length in blocks	00 01
10 – 11	Reserved	00 00

BYTE	FIELD	VALUE
12 – 15	Current state of Extended ComID	Enum {Invalid, Inactive, Issued, Associated}
16 – 25	Time of Allocation of ComID	10 byte format
26 – 35	Time of Expirty of ComID	10 byte format
36 – 45	Time since last reset of TPer	10 byte format
46 – BLK-SIZE	Reserved	zero

If the TPer does not support a real-time clock, the Time values in the Verify_ComID_Valid response shall be all zeroes. If the TPer supports a real-time clock, the fields that report the time shall use the following format:

Year (4 digits) – uinteger_2

Month (2 digits, 1-12) – uinteger_1

Day (2 digits, 1-31) – uinteger_1

Hour (2 digits, 0-23) – uinteger_1

Minute (2 digits, 0-59) – uinteger_1

Second (2 digits, 0-59) – uinteger_1

Fraction (number of milliseconds, 0-999) – uinteger_2

Reserved – uinteger_1 – 0x00

If the current state of the ComID is reported as unknown, only the current time or time since last reset of the TPer is valid in the data payload. If the ComID state is reported as valid, the time of expiry will be less than the current time.

3.3.5.6 Management Layer

Commands dealt with in this layer will be IF-SEND and IF-RECV with Protocol ID = 1 and with a valid active ComID.

This is the first layer that makes use of tokenized and packetized payloads. Communications in this layer occur between the TPer Session Manager (TSM) and the Host Session Manager (HSM). All communications happen within Control Sessions. There is exactly one Control Session for each ComID.

The Control Session associated with a particular ComID starts as soon as the ComID is issued, with a default amount of flow control credits. When the ComID is retired, the Control Session is terminated. The flow control for the Control Session is performed in the same manner as the flow control for Regular Sessions, with the difference that the communication is between the TSM and the HSM and these entities are responsible for the flow control.

In the case where several different sessions are simultaneously started on the same ComID, the packet headers for communications will have different SNs because the HSNs will be different. As far as flow control is concerned, only the TSN matters since the Control Sessions are identified by the TSN=`0x00 0x00 0x00 0x00` and the ComID.

One of the main tasks of this layer is to manage the startup of Regular Sessions. During this process, the TSM and the HSM will assign the TSN and the HSN that will compose the SN for the Session to be created.

When the process is initiated the HSM assigns an HSN (i.e. newHSN). The HSM has the opportunity to make sure newHSN is different from any other HSNs in use by other sessions managed by it, though this is not required.

This HSN is used in the header of the packets containing the `StartSession` method, and the SN for this packet would be SN = (TSN=`0x00 0x00 0x00 0x00`, HSN=myHSN). Once the TSM receives the `StartSession` method it assigns a TSN to the session. The TSM shall assign the TSN, newTSN, in such a way as to guarantee that TSNs are unique per ComID. The TSN assignment is returned to the application through the packet containing the `SyncSession` method invocation. At this point the Session Number for the new session has been established – newSN = (newTSN, newHSN).

Once the TSM processes the `StartSession` method and returns the `SyncSession` response, the Regular Session will be open for the case of sessions that do not require challenge-response. For sessions that require challenge-response, the Regular Session will be open when the TSM finishes processing the `StartTrustedSession` and sends out the `SyncTrustedSession` response.

3.3.5.7 Session Layer

In this layer all communications occur within a Regular Session.

3.4 SP Operation Descriptions

The following section highlights how tables and methods are managed.

3.4.1 General SP Guidelines

The Admin SP manages Templates, creates other SPs under issuance control, and maintains information about other SPs and the TPer as a whole. There shall be exactly one Admin SP on every TPer that has SPs or that can have SPs issued. If present, the Admin SP cannot be deleted or disabled.

Each SP is created by mixing one or more of the Templates identified in the Admin SP.

A Template includes the following:

1. Its name. Each Template must have a different name. The TCG shall never define a Template whose name begins with an underscore. Any templates defined by a manufacturer that are not TCG specified Templates shall have a name that begins with an underscore.
2. A set of table and method definitions. These definitions will be used to define the initial tables and methods of any instance of that Template. Any tables added by a manufacturer to a TCG defined Template shall begin with an underscore.
3. Optionally, a maximum instance count. At any time there can be no more than this number of SPs based on this Template instantiated within the TPer.

An SP includes the following:

1. Its name. Each SP must have a different name. The Admin SP has the reserved name **Admin**.
2. Its tables. Tables are stored in the access-protected, non-volatile storage area on the TPer.
3. A set of methods.

All SPs must be created from at least the Base Template. The Base Template may be combined with any other Template(s) to create an SP, though the number of SPs that instantiate a particular Template may be limited.

3.4.2 Access Control

Access Control limits the methods that can be executed on an SP, a table, or on specific rows and columns of a table.

Permission to execute a method is governed by which secrets the method's invoker has proven that it knows. The secrets and their public parts are called **Credentials**. The operation for proving knowledge of a secret is called an **Authentication Operation**. The actual proving of knowledge of a secret is called **Authentication**.

Authentication in this document may be described as either **Explicit Authentication**, which occurs as a result of challenge/response, for example; and **Implicit Authentication**, which occurs as a result of implicitly proving knowledge of a secret, such as during session key exchange. An authority is considered authenticated in either type of scenario - the terms Explicit and Implicit are descriptive and do not limit the authentication or capabilities of an authority.

In addition to authentication, credentials may also be used for **Encryption**.

The `Authority` table on an SP associates specific Credential-Operation pairs together in **Authority** objects. For example, one authority may include a credential that contains the secret password with retry and hiding specification and the stipulated proof-of-knowledge operation of password authentication.

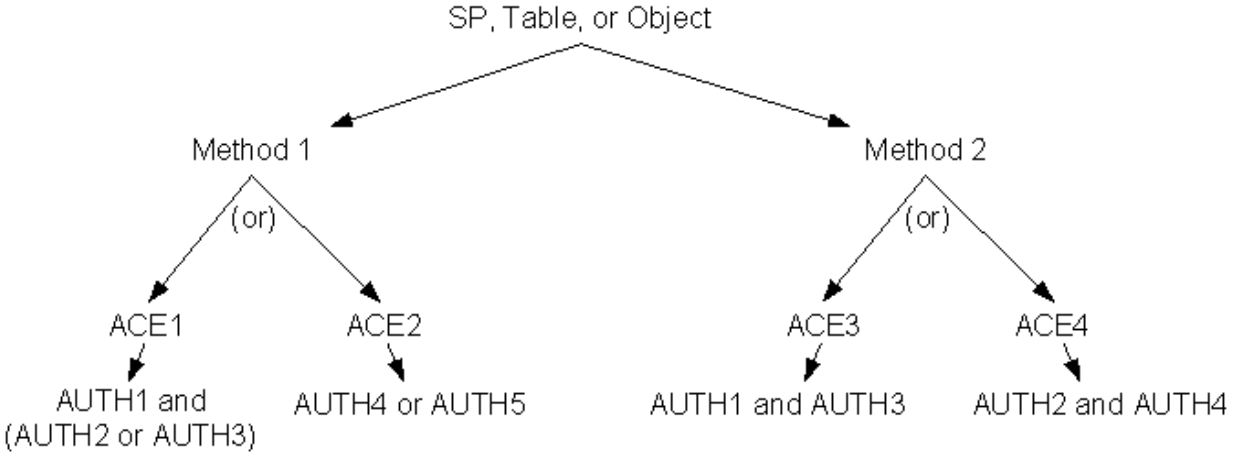
An authority can be used by the host application to represent a person, a role, a program agent, etc. These are distinctions of meaning to the application, not to the SP.

Certain authorities are defined by this specification. The AdminExch authority, of the class Admins, is one such pre-defined authority. Every SP has an AdminExch authority at time of issuance. Security Subsystem Classes may specify authorities in addition to these, or may restrict the use of some of these. For details regarding the Admin authority and other pre-defined authorities, refer to 5.3.4.1.2.

Access Control is specified in layers. The top layer of the mechanism is Access Control Lists (ACLs). ACLs are lists of **Access Control Elements (ACEs)**. This layering gives the host a way in which it can delegate control of an ACL, via control of its ACEs, to various independent entities.

ACEs are Boolean combinations of authorities. This permits the ACE to express cross-certification or other forms of restriction. When an Authority is authenticated, it is True in the Boolean expression, and False otherwise.

Figure 10 Access Control



An authority may be one of two kinds: Individual and Class. Each Individual Authority may be a member of one Class Authority. A Class Authority may also be a member of one Class Authority. A **Class Authority** is identified only by its Authority UID and Class Name. A Class Authority does not refer directly to a Credential. An **Individual Authority** specifies one Credential and one Operation on the Credential.

A Credential is an object in a `Credential` Table. All credential tables have a name that starts with "C_". A credential table must have at least one column for a secret. It may also have "public" parts,

such as public keys and certificates. A particular credential need not have all its columns filled in. For example, if only a public key and certificates validating that public key are known, then the private key columns may be unused (zeroes in these columns indicate that this information is not present).

A Credential Table must also have internal implementation for using the secrets and the "public" parts of each credential and must handle all optional parts.

The operation is selected, as appropriate for the credential, from:

- Password or PIN or Passcode
- Signing:
    - Public Key Challenge/Response Sign/Verify
    - Symmetric Key Challenge/Response Sign/Verify
    - HMAC Challenge/Response Sign/Verify
- Key Exchange (Certificates or other methods provide implicit Authentication)
    - Public Key Encrypt/Decrypt
    - Symmetric Key Encrypt/Decrypt
- None or "" This operation will always succeed and therefore the authority will always Authenticate.

A Class Authority is authenticated when an Individual Authority that is a member of that Class Authority is authenticated. Class Authorities cannot be directly authenticated. Class Authorities are a convenient way to allow an ACE to be set on a method without enumerating all the Individual Authorities that may authorize that method. This means that the Individual Authorities that belong to that Class Authority can be changed without having to change any of the ACEs that refer to the Class Authority.

Access control is permitted in that ACEs can apply to methods on an SP, on a particular table in an SP, or on arbitrary parts of a particular table in an SP, down to the granularity of a single table cell. Note that access control over reading the columns that participate in an object table's index gate whether or not that object may be read.

With ACEs as the building blocks of ACLs, each ACE can have separate managerial control. For example, one authority might create a table and give another authority control of some of the ACEs on that table. This allows flexible, fine-grained management of access.

The simplest ACL is one ACE of one authority. The minimum and maximum number of ACEs in an ACL and the minimum and maximum number of authorities in an ACE is Security Subsystem Class-specific. Every Security Subsystem Class shall at least stipulate the minimum.

Authentication to an authority occurs within a session (or during session startup) and applies only to that session. All authorities that participate in successful session startup are authenticated for that session. During a session the host may make any number of `Authenticate` method invocations. There may be Security Subsystem Class-defined TPer and per session limits on the maximum number of authorities that may be authenticated at any one time.

Security is enhanced by logging events that are related to ACLs. Authorities determine when attempts to use them are to be logged (authentication failures, etc.).

3.4.3 SP Issuance, Personalization, and Operational State

Issuance is the cryptographically controlled creation of SPs from Templates. Issuance occurs within a session to a TPer's Admin SP, and is achieved by demonstrating knowledge of the secrets required to authorize the creation of new SPs and then, for each new SP, creating a unique credential for the Admin authority on that SP. Issuance is not considered complete until the Issuance session to the

Admin SP successfully closes. Templates can not be included in an SP except during the initial issuance of that SP.

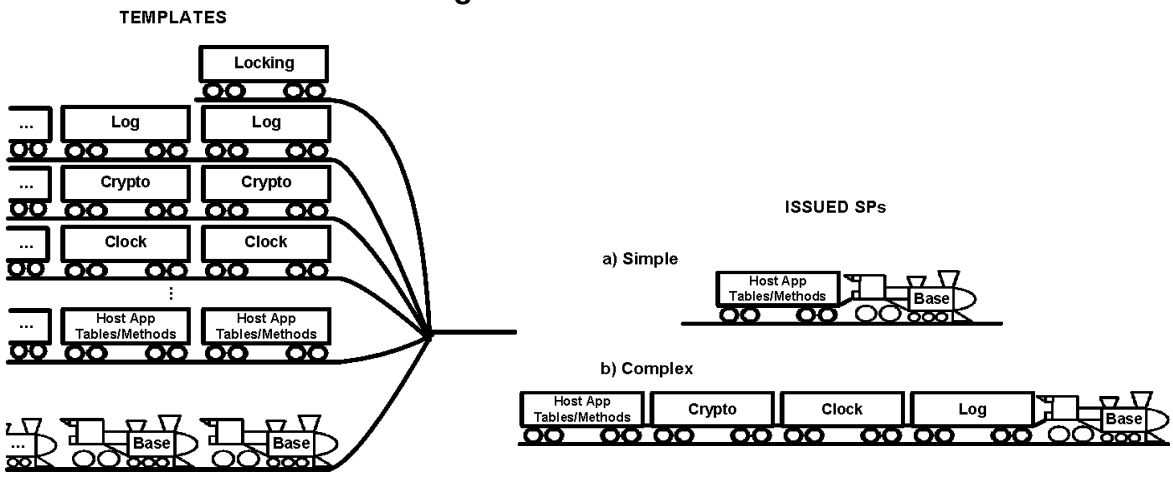
Personalization follows Issuance. The Admin authority on the new SP can accomplish personalization by opening a session to the issued SP, creating new tables and methods (in addition to the tables and methods that were provided by the Templates), provisioning those tables, and, finally, setting the access controls on the SP's methods. Personalization will not be considered complete until the session is successfully closed.

Operational State is reached once personalization is complete. The result of the Issuance and Personalization process is an SP both usable by and useful to a host application.

3.4.3.1 Example – Issuing an SP

Issuing an SP is similar to building a train (see Figure 11 below). Every train (SP) must have an engine (Base Template). Additional cars (other Templates) providing additional capabilities may be added at the time of issuance. In the simplest case, an SP is issued from just the Base Template (see part 'a'). In more complex cases several Templates can be used.

Figure 11 Issuance



3.4.4 Sessions, Methods, and Transactions

All communications with an SP occur within sessions. A session is always started by a host.

Normally the host application will end a session when it has finished its communication, but either the SP or the host may abort a session at any time for any reason. For a specific SP there may be any number of Read-Only sessions active simultaneously, but only one Read-Write session. Read-Only and Read-Write sessions are mutually exclusive. The existence of Read-Only sessions, the maximum number of simultaneous Read-Only sessions that may be opened to any SP, and/or limit the total number of open sessions available to a TPer shall be defined by Security Subsystem Class.

Except as noted in the SP Reference, no explicit changes to an SP made during a Read-Only session are made permanent, even when the session closes successfully. Indirect changes, such as PIN blocking, log updates, etc. will remain persistent.

Methods are procedures that operate on tables or SPs, and are called within a session to an SP. The caller passes a list of parameter values to the method and the method returns a list of result values followed by a status list, the first value of which is the status code response to the method invocation. Status code 0 (OK) means the method call completed successfully. Failure conditions are assigned specific non-zero status codes (see Status Codes in 5.1.3 for details). Within a given session at most one method shall be active at a time.

Method calls, their parameters, and their results are all sent and received over session streams. Each session to an SP has at least two streams of bytes onto which data is encoded. One stream goes from the host to the SP, and the other comes from the SP to the host. Each stream operates asynchronously from all other streams.

Typical host method calls will send all their parameters/data to the SP before trying to read any of the results, but the SP is free to generate results incrementally as it consumes its parameters. The host is similarly free to try to read SP results while sending parameters. The SP implementation decides how synchronous or asynchronous to be, so long as the semantics of the method call(s) are not compromised.

Transactions are used to provide a clean model for how changes to an SP are to take effect. They also provide an easy way for host applications to handle error recovery. If a session is aborted, any open transactions are aborted.

Changes are successfully committed and made persistent (to the media, made visible to subsequent sessions on the same SP, etc.) in 2 ways:

1. When a method is invoked outside of a transaction, and resolves successfully, changes made by that method are committed and made persistent immediately.
2. When a method is invoked inside of a transaction or set of nested transactions, changes made by that method are committed and made persistent when the top-level transaction closes.

Effects on other aspects of the TPer (i.e. hardware settings) occur when associated changes are successfully committed. Note that some changes occur as exceptions to transactional control (i.e. logging), and commit immediately even if they occur inside of a transaction or following a method invocation that has failed.

3.4.4.1 Method Calls

A method call consists of the following steps:

1 The host tells the SP the method it wants to call.
2 The host sends a list of parameters to the SP.
3 The body of the method is executed in the SP.
4 The method results are returned from the SP to the host.
5 Steps 2-4 may be repeated when input and output are incrementally streamed.

3.4.4.2 Transactions

In addition to method calls, a session may include nested transactions. The maximum number of transactions that can be nested is Security Subsystem Class-specific, and will be specified in response to the `Properties` method invocation. There are tokens in the session stream that are transaction start and end tokens (defined in the 3.2.3). The host is free to start a transaction on a session between method invocations, and may subsequently open nested transactions. All transactions consist of the following steps:

1. The transaction is opened.
2. A set of method calls is made.
3. The transaction is either aborted or committed.

If a transaction is aborted all SP state is reset ("rolled-back") to its value at the time the transaction was opened.

Nested transactions abort or commit relative to their parent transaction. In the case of an aborted transaction, the SP state is rolled back to the point where the transaction was started. (This is true whether or not the transaction is nested.) In the case of a commit, the nested transaction's changes

become part of its parent transaction, as if the nested transaction boundaries had never been established.

A commit of a nested transaction does not make a commit that necessarily persists since the parent transaction is not yet ended. All transactions must be closed before data is written to the SP.

The TPer must guarantee that a transaction completely commits to media (persists) or completely aborts. This means that the TPer must arrange that if a power cycle, reset, or other event, occurs in the middle of a commit, that when the TPer comes back up the commit is either finished or all the changes are aborted. This guarantees SP consistency and prevents power-off or reset attacks.

3.4.4.3 Session Manager Protocol Layer

The Session Manager is a special protocol layer session on any TPer with SPs. It is used by host applications to start and stop sessions with SPs and to inquire about overall TPer communication characteristics. The Session Manager protocol layer does not provide a session "to" any SP – it is a communications control session. This session is always open and attempts to close it will always fail. The method calls available on the Session Manager layer are identified in section 5.2, and include `Properties, StartSession` and so on.

Although method invocations on the Session Manager layer cannot change permanent state on the TPer, some method invocations may have side effects that occur outside of the normal method invocation process, such as logging or PIN retry counts. In cases where these changes should occur – for example, logging a `StartSession` method call success or failure – the change occurs on the SP to which the method call was attempted.

Method calls on the Session Manager layer are formatted/encoded the same as on any other session. Due to the asynchronous nature of session startup and TPer communications, all of Session Manager layer methods' responses are formatted as method calls, so that the host may identify responses to methods it has invoked.

For more information on protocol layers, and the Session Manager layer in particular, reference section 3.3.4

3.4.4.4 Ending Sessions

The Host or TPer is free at any time to end a session in which it is participating, but only the host shall end the session successfully.

The session is not considered successfully closed until the party receiving the end of session request has responded indicating whether or not it was able to comply with the session ending request. Thus, a session is successfully ended when the TPer receives an End of Session token (see section 3.2.3.2) from the host and responds with an End of Session token, and when flow control for ending the session has been performed as noted in Section 3.4.6.5.

When a session closes, TPer resources that had been reserved for use with that session are released. The release of resources is not dependent on whether the session closed successfully or unsuccessfully – the end of the session releases the resources.

Sessions may end unsuccessfully (abort) in a number of ways. These include (but are not limited to):

- If the TPer detects any violation of flow control.
- The host sends an End of Session token, but does not receive a response from the TPer within the host's timeout period. In this example, the TPer would also time out while waiting from some response from the host.
- If the host does not (or can not) send an End of Session token to the TPer, and sends no other communications, the TPer would time out while waiting for the communication from the host.

- If a timeout by one or both of the communicators in a session timeout before flow control acknowledgements are received.
- If a negative acknowledgement is received by a communicator on one of the End of Session subpackets. This session has not yet ended, and may still end successfully through retransmission of the End of Session token. Alternatively, the communicators may timeout waiting for communications.

If a session is ended in the middle of the transmission of a method call or its parameters, then the method call is aborted in addition to the session being aborted. This is considered a fatal session error indicating a communication synchronization error (or worse).

When a session is aborted, open transactions within that session are also aborted.

The TPer may send a `CloseSession` method on the Session Manager layer when it aborts a session. This is done by the TPer to notify the host that the TPer is ending the session.

Hardware resets and power cycles cause all open sessions to abort.

3.4.4.5 Starting Sessions

Starting a session depends upon three independent requirements:

1. The TPer and the requested SP having sufficient resources.
2. Negotiating symmetric keys if secure messaging is required.
3. Authenticating requirements (one of the following):
    a. Host must authenticate to SP
    b. SP must authenticate to Host
    c. Both of the above
    d. None of the above (No authentication)

The first requirement, sufficient resources, is often time-dependent, so if a session fails to start for this reason a short delay may be necessary before retrying.

The host sets the second and third requirements when it attempts to start the session, as described below.

Sessions are started with either a two or four method exchange on the Session Manager protocol layer:

`StartSession`
`SyncSession`
`StartTrustedSession` (optional)
`SyncTrustedSession` (required if `StartTrustedSession` is used)

Note: Because of the asynchronous nature of session startup and other Session Manager layer traffic, the `StartSession`/`StartTrustedSession` responses (`SyncSession`/`SyncTrustedSession`, respectively) are formatted as a method call back to the host. Host and SP are the relevant side's session numbers (if the session successfully starts).

The Host application starting the session determines the secure messaging and authentication requirements to be satisfied by specifying up to four authorities:

- HostExchangeAuthority: Host's Exchange Key – used for exchange of session keys, provides implicit authentication
- HostSigningAuthority: Host's Signing Key – used for authenticating the host to the SP and session startup method integrity, provides explicit authentication

- SPExchangeAuthority: SP's Exchange Key – used for exchange of session keys, provides implicit authentication
- SPSigningAuthority: SP's Signing Key – used for authenticating the SP to the host and session startup method integrity, provides explicit authentication

Note: These authorities are already known to the SP.

Host authorities, if used, are passed in the `StartSession` method call. SP authorities are bilateral authorities called out in the Host authorities' `Authority` table rows. The ability to specify authorities in the `StartSession` method call, coupled with the linking of authorities in the `Authority` table, provides a large and diverse set of possible session protocols, including secure messaging. It is the initial selection of authorities by the host that determines which protocol is to be followed.

Note: When the host makes the `StartSession` method call it knows which SPExchangeAuthority and SPSigningAuthority (if any) the SP will use. Those may be the root authorities in a certificate chain whose ultimate effective authority the host does not know. This is why the SP may return certificates to the host as part of `SyncSession`.

If a HostSigningAuthority or SPSigningAuthority requires a Challenge-Response, as is the case for all PuK, SymK, and HMAC authorities, or if secure messaging is to be used (or both), then the `StartSession` and `SyncSession` methods will be followed immediately by the `StartTrustedSession` and `SyncTrustedSession` methods.

An authority (HostExchangeAuthority, SPExchangeAuthority, HostSigningAuthority, or SPSigningAuthority) that is also a Public Key Authority (an Authority with public key credentials--PuK) may have additional information supplied for it in the form of a certificate or certificate chain. In this case the Effective Authority (the one responding to the challenge) will be the tail PuK of that chain. The effective authority is transient to the session. When an effective authority is transmitted to the SP, the full contents of its certificate chain will be available only during the session. It is necessary to create a new authority on the SP (in a Read-Write session) if the host wants that authority to persist on the SP past the end of that session.

All authorities that participate in the successful startup of a session are authenticated for that session.

3.4.4.6 Session Timeouts

A session timeout is associated with every session and is specified in milliseconds. The session timeout can be used to limit the lifetime of a session. A value of zero indicates that the timeout value is infinite, in effect disabling the timeout feature.

The session timeout is a property of the session and is derived from three sources.

1. DefaultSessionTimeout : A mandatory value in the `Properties` method response.
2. SPSessionTimeout : A mandatory column in the `SPInfo` table.
3. SessionTimeout : An optional parameter in the `StartSession` method call used to open the session to the SP.

The TPer may impose further conditions on maximum and minimum timeouts supported by the device depending on hardware and other design considerations. These will be indicated in the `Properties` method response values MaxSessionTimeout and MinSessionTimeout. These limits will apply to all of the three timeouts listed above.

The actual timeout value used by the TPer device for connection will be the one that corresponds to the shortest of the above three timeout values.

A row in the `SPInfo` table contains the SP suggested default timeout SPSessionTimeout and can be modified by the host if it has appropriate authority. This value will take effect immediately on all sessions open on the SP including the session that made the change itself.

The TPer and the Host both maintain a timer associated with every allocated active session. The timer starts when a session is successfully opened to an SP. Depending on the type of session started, this

would be when the tokens for the SyncSession or the SyncTrustedSession method call is built by the TPer and made available to the host.

The timeout does not apply to the Session Manager layer since it is always open. The time taken to complete the Session Manager layer exchange to successfully start a session is not in the scope of this feature.

If, at the end of the timeout, the session has not been terminated, the TPer shall abort the session. The session is considered to have been closed / terminated when the last status token sent by the TPer is picked out from the output buffer by the host, or when the TPer releases all the resources (including the output buffer) for the session.

3.4.4.7 Signed Hashing During Session Startup

If a Signing Authority is invoked in a session startup method (for either the Host or the SP), and the authority's HashAndSign column indicates that hashing is required, the signing Credential referenced in that Signing Authority's row of the Authority table and the hash protocol identified in the Hash column of the associated credential are used for the hash/sign operation on the session startup methods.

Session startup shall fail if an authority indicates that hashing and signing the session startup methods are required and does not include the signed hash as a parameter of the method invocation.

If a Signing Authority requires the hash/sign operation to be performed, that Authority's row of the Authority table shall indicate an Operation/Credential pair of Signing/Private Key, SymK/Symmetric Key, or HMAC/Symmetric Key.

The signed hash is sent as the last parameter of the method call and hashes the entire method call (except the hash).

Note that the HostSigningAuthority and SPSigningAuthority provide separate controls for hashing/signing method invocations from the host to the SP and from the SP to the host, respectively. This means that hashing and signing may be performed in one of the two communications directions, in both directions, or in neither direction, depending on the HashAndSign column values of the HostSigningAuthority and SPSigningAuthority.

3.4.5 Session Examples

Seven examples from among the many possible ways to start a session:

(Protocol diagrams for each one are provided below)

- **None.** No Authorities are used. This is a non-authenticated, non-secure messaging session. (Reminder: The Anybody Authority is always authenticated.)
- **Host PIN.** This is the rudimentary case of passcode authentication, where the passcode is passed in the clear. Secure messaging is not an option in this case.
- **SP Symmetric Key Exchange.** The simplest case that provides for full Host & SP session key encryption. The SP needs to perform only symmetric encryption.
- **Full Public Key.** This uses public keys for signing and key exchange, for both the Host application and the SP. With a proper certificate chain or other validation proof for the exchange key, this is also authenticated. SP Issuance is an example where Full Public Key is used.
- **Full Symmetric Key.** This uses symmetric keys for signing and key exchange, for both the Host application and the SP.
- **Host Public Key Authentication.** This is a simple, strong enabler that does not start up secure messaging. An example use case might be a TCG TPM that authenticates a session in order to unlock the read/write functions of a disk drive and, because of the nonce and the private key, does not need a secure channel.

- **Host Public Key – SP Symmetric Key.** This is a case where it is desired that the SP sign, but that public key signing, and indeed all the private key operations of public key cryptography, are deemed too computationally expensive for the SP. The Host application is allowed to perform private key signing and the SP to perform public key verification and public key to encrypt a session key. The SP only symmetric key signs, and does symmetric session key receipt.

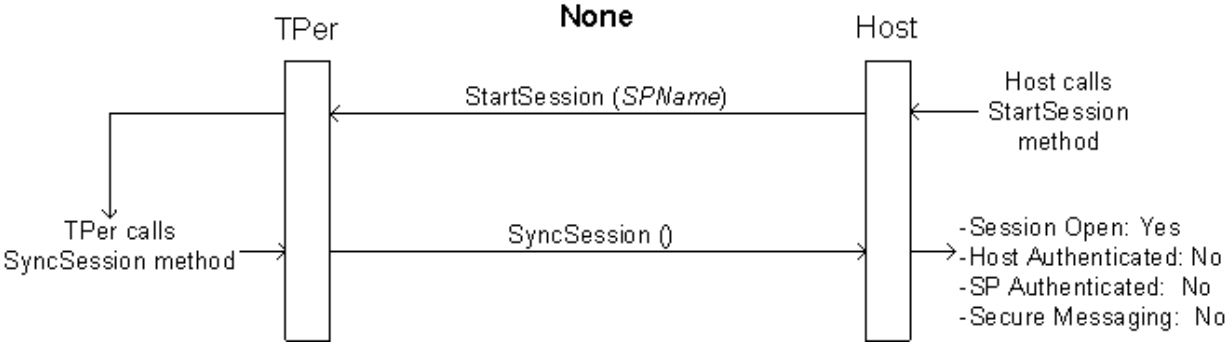
Descriptions of the usage of these authorities and protocols are found in section 5.3.4.

NOTE: For clarity, only the security related parameters are shown in these diagrams.

3.4.5.1 No Authority Example

For this example, no authorities are invoked during session startup. This session startup protocol results in startup of a session that permits only actions by the Anybody authority.

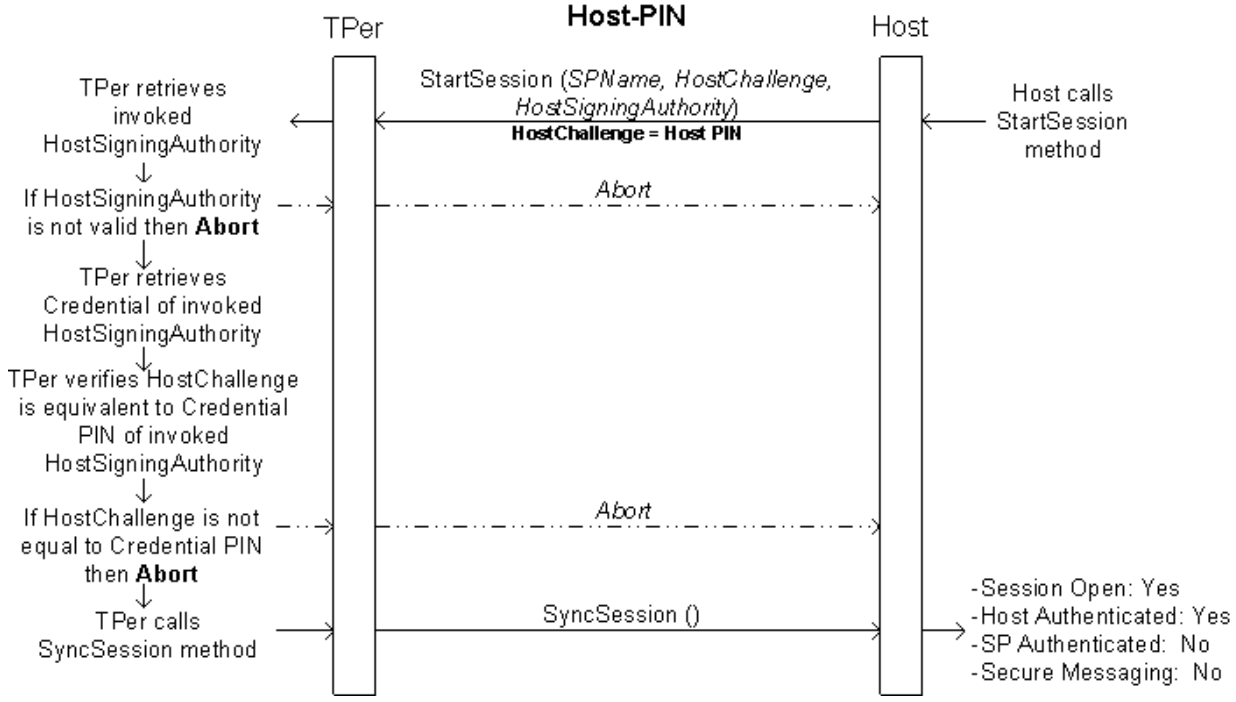
Figure 12 No Authorities Used



3.4.5.2 Password Example

This example displays the use of a Host Signing Authority to perform session startup. The referenced Host Signing Authority has an `Operation` column value of Password. The `StartSession` method invocation transmits a PIN as the value of the HostChallenge parameter. The TPer validates the value of the HostChallenge parameter to the value of the `Password` column of the `C_PIN` credential object referenced by the Host Signing Authority. Successful session startup results in authentication of the Host Signing Authority.

Figure 13 Pass Code Authentication
Host-PIN



3.4.5.3 Full Host & SP Session Key Example

The session startup example detailed in this section involves invocation of the `StartSession` method by the host with an authority referenced in the HostSigningAuthority parameter.

The `Operation` column value of the authority referenced as the HostSigningAuthority is None. This indicates that this authority will not perform a challenge-response for this session startup method exchange.

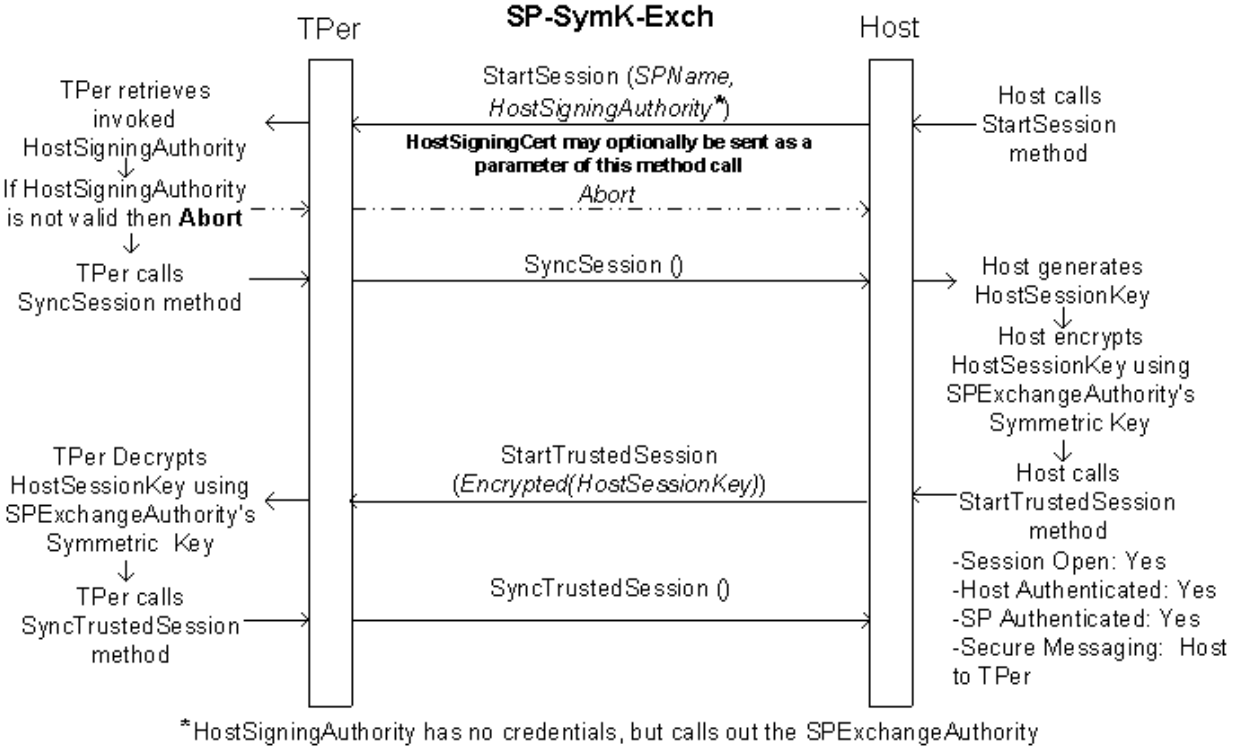
The value of the `Secure` column of the HostSigningAuthority defines the type of secure messaging required. In this case, the HostSigningAuthority's `Secure` column identifies that only confidential messaging is required.

In this example, the authority invoked by the host as the HostSigningAuthority references an SP Exchange Authority in the HostSigningAuthority's `RespExch` column. This authority's credential will be used to encrypt the session key used for confidential messaging. This credential could be either a public key or a symmetric key. In this example, the credential is a symmetric key.

The SP Exchange Authority does not require secure messaging in this example (i.e., the SP Exchange Authority's `Secure` column value is None), so only messages sent from the host to the SP will be encrypted.

Successful session startup results in implicit authentication of the Host and the SP.

Figure 14 Host Session Key Encryption



3.4.5.4 Host Public Key Authentication Example

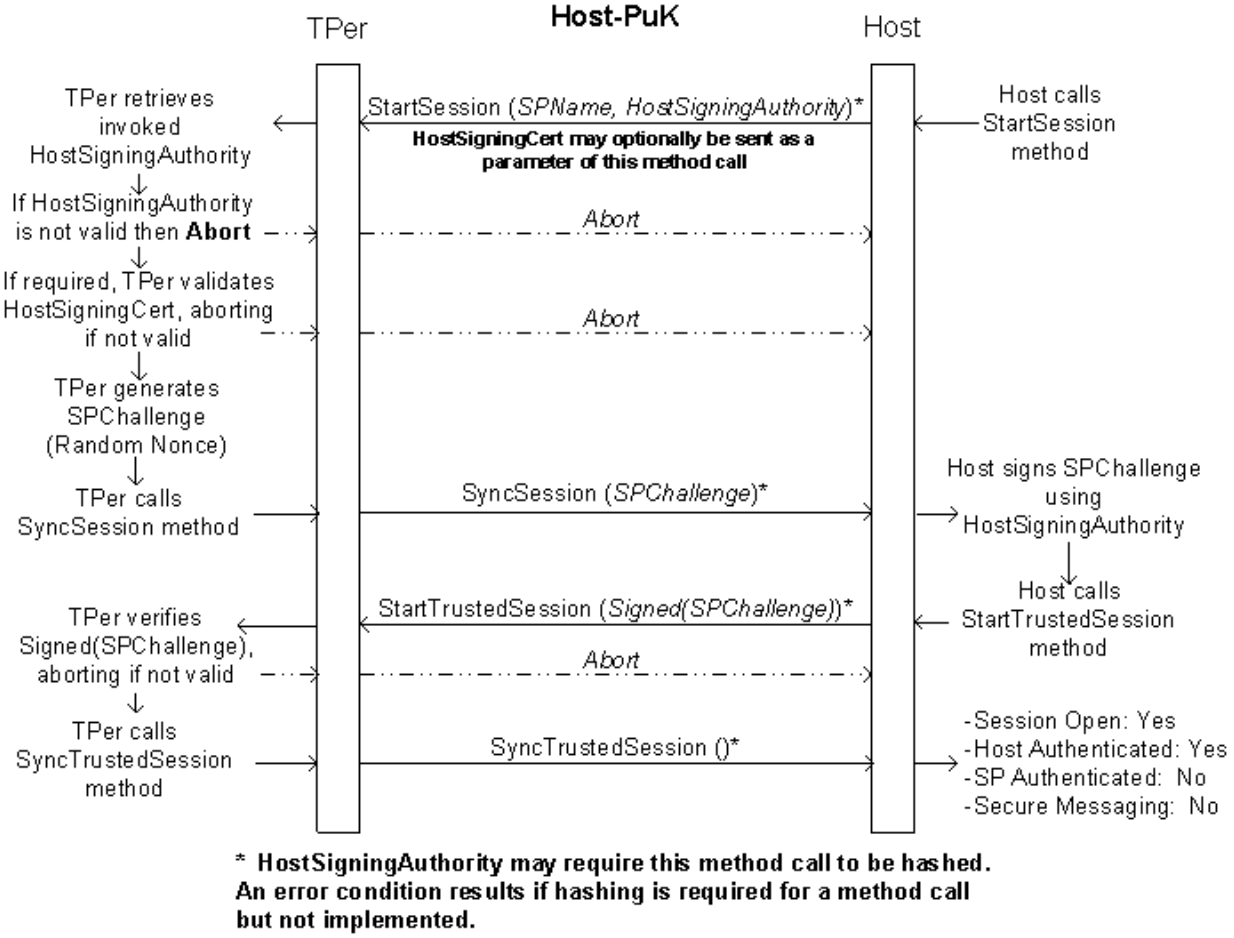
The example in this section identifies a session startup protocol wherein the host invokes the `StartSession` method with an authority value in the HostSigningAuthority parameter. For this example, the invoked HostSigningAuthority has an `Operation` column value of Sign. This indicates that the authority references a public-private key pair.

The TPer responds to the `StartSession` invocation with a `SyncSession` method invocation that includes a value in the SPChallenge parameter. The host signs the SPChallenge parameter using the private key of the public-private key pair it is using for this session. The host then invokes the `StartTrustedSession` method and passes the signed SPChallenge value as the StartTrustedSession's HostResponse parameter value.

The TPer validates the signed response using the HostSigningAuthority's public key. If the validation is successful, the session starts.

Successful session startup results in explicit authentication of the HostSigningAuthority (through the challenge-response mechanism).

Figure 15 Host Public Key Authentication



3.4.5.5 Full Public/Symmetric Key Examples

In this example, the host invokes the `StartSession` method with authorities referenced in both the HostSigningAuthority and HostExchangeAuthority parameters. The HostSigningAuthority references authorities in its `RespSign` and `RespExch` column values, which are used as the SP Signing Authority and SP Exchange Authority respectively.

The host also includes in the `StartSession` method invocation a value in the HostChallenge parameter.

Upon receipt of the `StartSession` method, the TPer returns the `SyncSession` method with a random value in the `SyncSession` method's SPChallenge parameter.

The host signs the SPChallenge parameter with the HostSigningAuthority's referenced credential – this could be an HMAC if the HostSigningAuthority has an `Operation` column value of "HMAC" and references an HMAC credential or a private key encryption if the HostSigningAuthority has an `Operation` column value of "Sign" references a public-private key pair.

In addition, in this example the HostSigningAuthority's `Secure` column value requires confidential messaging. The host generates a session key for encryption of the messages the host will be sending to the TPer. The host encrypts this key with the SPExchangeAuthority's credential (this could either by

a symmetric encryption or a public key encryption, depending on the type of credential referenced by the SPExchangeAuthority).

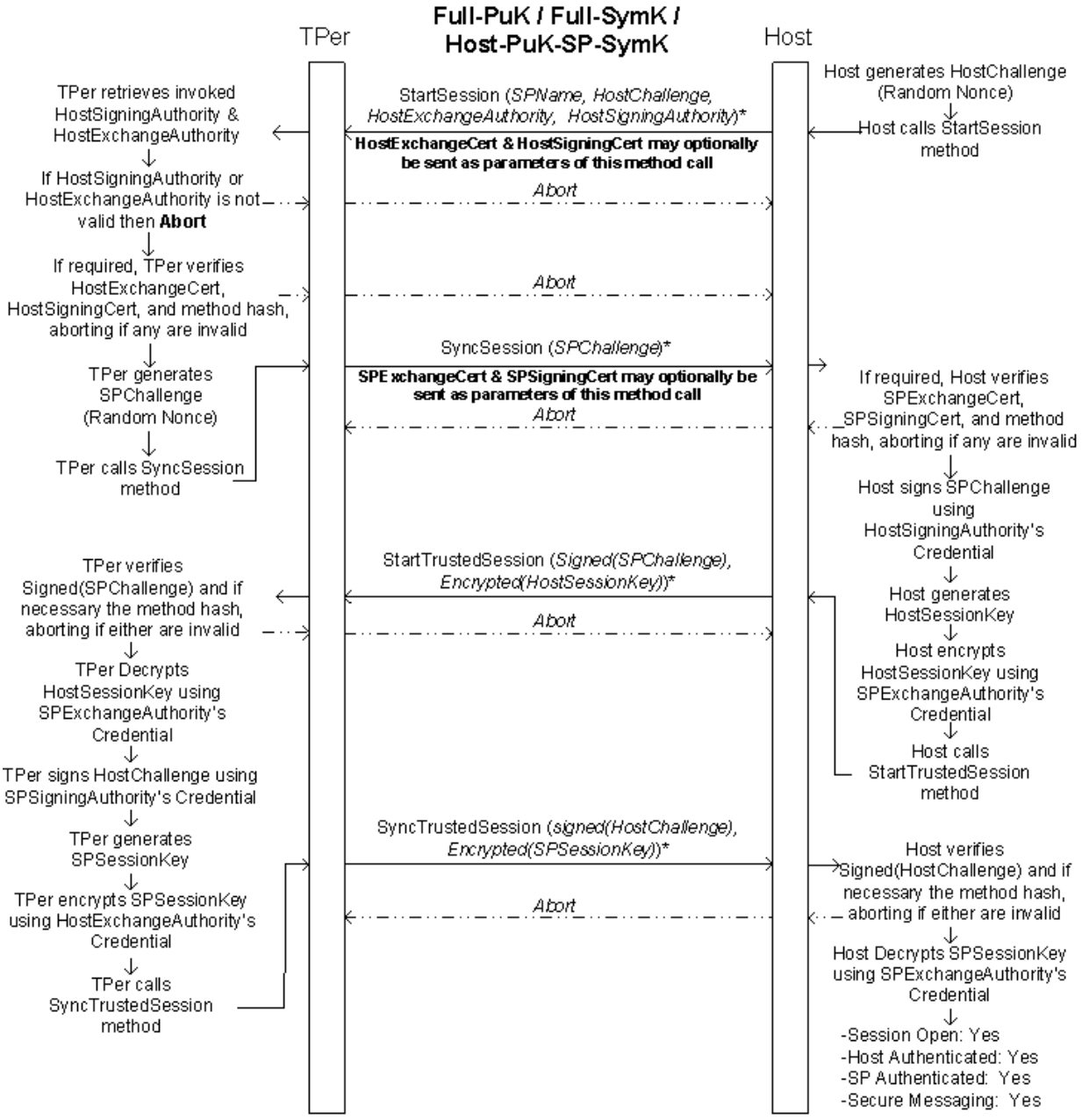
The host then invokes the `StartTrustedSession` method, passing the encrypted session key and the signed SPChallenge as parameters.

The TPer verifies the signed SPChallenge, signs the HostChallenge value, and generates a session key and encrypts it with the HostExchangeAuthority's credential.

The TPer returns the `SyncTrustedSession` method with the encrypted session key and the signed HostChallenge as parameters.

Upon verification of the signed challenge by the host, the session may begin. The successful session startup results in explicit authentication of both the HostSigningAuthority and the SPSigningAuthority, and the implicit authentication of the HostExchangeAuthority and SPExchangeAuthority. All messages sent by each communicator will be encrypted.

Figure 16 Full Public Key, Full Symmetric Key, and Public/Symmetric Key Authentication



Full-PuK - All Authorities call out Public Key Credentials
Full-SymK - All Authorities call out SymK Credentials
Host-PuK-SP-SymK - Host Authorities call out Public Key Credentials, SP Authorities call out SymK Credentials

*** Any invoked Signing Authority may require method call hashing.**

3.4.6 Stream Flow Control: Host & TPer

Flow control ensures that when data is sent from a source to a destination that the destination has enough buffer space to receive it. There are two kinds of flow control: Interface and Stream data.

Interface flow control is involved in moving T10 SECURITY PROTOCOL IN/OUT or T13 TRUSTED SEND/RECEIVE commands across an interface between a host and TPer, and is outside the scope of this document.

Stream data flow control is used to keep a Host or TPer from overwhelming the other party with data during a session. All session streams have flow control. Flow control violation is one reason either communicator may abort a session.

3.4.6.1 Transmission Acknowledgement

For an SSC that supports transmission acknowledgement, each packet sent from the TPer to the Host (or vice-versa) for a given session has a sequence number (SeqNumber) that corresponds to the number of packets that have been sent by that communicator since the start of the session. The first packet in a session shall have a SeqNumber value of 1.

If transmission acknowledgement is supported, each packet with SeqNumber N must be acknowledged by the receiver. Once the sender receives an acknowledgement for data contained in packets up to packet N, the sender may safely discard the data for packets with SeqNumber N and lower.

Packets that contain only ACK or NAK information, and no session data, shall not require an ACK/NAK response from the receiver. These packets shall still have an appropriate SeqNumber field value. Packets for sessions that are not protected by secure messaging that do not require ACK/NAK shall be those packets with a Length field value of zero and a corresponding empty Data field value. For packets that are protected by secure messaging that do not require ACK/NAK shall be those packets with a DataLength field value of zero and an empty Data field (The IV and MAC fields may still contain values).

3.4.6.2 Transmission Negative Acknowledgement

If the receiver detects missing or invalid data, the receiver shall send a negative-acknowledgement packet (NAK) with the SeqNumber of the packet at which the receiver wishes the sender to begin retransmission. Generally, the receiver will put a value of the SeqNumber of the last known good packet (N) received plus one (this automatically acknowledges all previous packets with SeqNumbers less than equal to N). The communicator shall not NAK a SeqNumber less than or equal to the last ACKed SeqNumber. Negative acknowledgement serves to notify the sender that a retransmission of packet N+1, etc. is needed.

[ms2]Upon dispatch of the NAK, the sender of the NAK shall discard all packets with SeqNumbers N+ 1 and higher, since the sender will retransmit these. After the first NAK is sent for packet N+1, all further data shall be discarded until packet N[ms3] is received. Retransmission of the NAK is dependent on the transmission timeout value for the session, not on subsequent receipt of additional data.

3.4.6.3 Transmission Timeouts

The flow control timeout is set during the exchange of session startup methods `StartSession` and `SyncSession`. The flow control timeout for a session takes effect after session startup has successfully completed. Both communicators share the same transmission timeout value.

The sender may provide, in the `StartSession` method, a value for the TransTimeout parameter. The communicator that transmits the `SyncSession` method may include a value for the TransTimeout parameter. If so, that communicator's timeout value shall be larger than the `StartSession` TransTimeout value. In either case, the TransTimeout value shall be greater than or equal to the MinTransTimeout value and smaller than or equal to the MaxTransTimeout parameter reported in the `Properties` method response. If neither communicator includes a value for the TransTimeout parameter, the DefTransTimeout value, as reported in the `Properties` method response, shall be used.

If the sender detects a missing acknowledgment by means of a timeout, the sender shall retransmit the data from the last valid acknowledgment. If the sender still receives no acknowledgement after a timeout period, the sender shall re-transmit the same packet or the same packet with added data if more data is available. This retransmission will repeat up to an implementation-specific number of times. Thereafter, the sender shall terminate the session, i.e. no more data can be transmitted for this session (the session will timeout at some point and be closed by the receiver).

3.4.6.4 Buffer Management

Flow control is used to keep a Host or TPer from overwhelming the other party with data during a session. Violating flow control is one reason either side may abort a session.

Before session data can be sent, the destination needs to notify the source that it is ready to receive data and how much data it is able to receive. This is done by sending a Credit Control Subpacket in the direction opposite that of the data.

The InitialCredit parameters of the `StartSession` and `SyncSession` methods provide each communicator in a session the opportunity to provide an initial amount of credits for use when the session successfully starts. If either of these values is omitted, then once a session has been successfully started, the communicator that omitted the value from the InitialCredit parameter of its session startup method shall send to the other communicator a credit subpacket announcing its available session buffer space.

The exchange of credits permits data to be moved from one communicator to the other. As data in the receive buffers of the communicators is consumed and space released, additional credit control subpackets may be sent.

The sender shall not send more data than it has credits from the receiver. As the sender transmits data, the amount of transmitted data is subtracted from the total credits that had been provided to the sender. This identifies the amount of data that may still be sent without receiving additional buffer credits.

As the receiver consumes data, the receiver may notify the sender that additional data may be sent. This is done by transmitting a credit control subpacket identifying how much additional buffer space the sender may utilize. The data sender can then calculate how much more data may be sent. That is, the number of bytes of data that can be sent to that session will be increased by the value of each credit received. When a communicator transmits data, the amount of data sent is subtracted from the credit total.

For SSCs that support flow control, credit subpackets are required after ComID acquisition, so that the host and TPer may exchange methods/responses on the Session Manager Layer. This credit only applies to the Session Manager Layer for that ComID. Credit subpackets are also required immediately after session startup (unless values are posted in the InitialCredit parameters, in which case additional credit subpackets are optional at this time).

Otherwise, credit control subpackets should be sent infrequently and be bundled with other traffic, in order to minimize interface overhead. Either communicator in a session may send credit subpackets as frequently as in every packet, or when a threshold is reached (e.g. the unreported credit is more than some percentage of the buffer size).

Credit values are byte counts for data only. They do not include packet or subpacket headers/overheads.

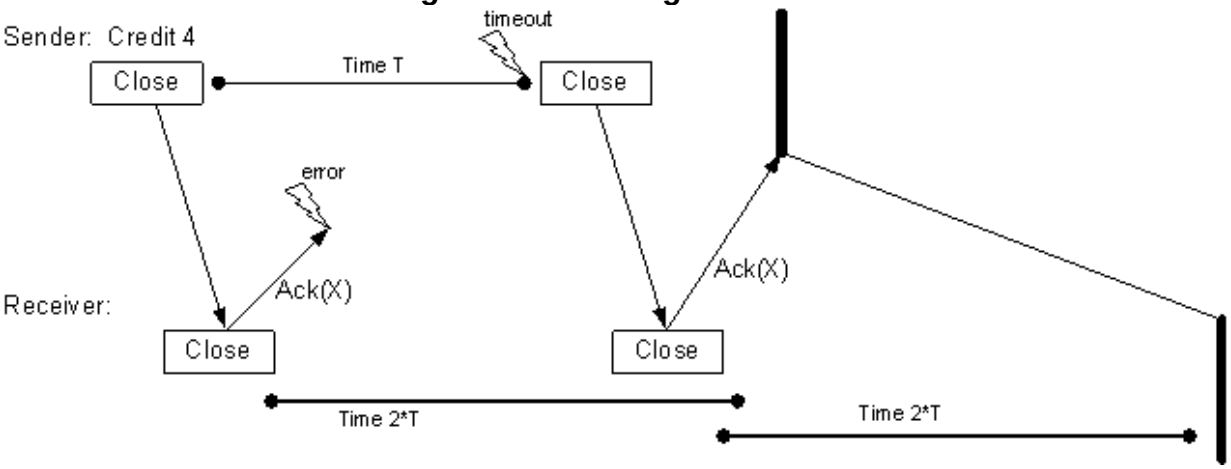
3.4.6.5 Closing a Session

When the sender transmits a packet that contains a close session control subpacket (which is a subpacket in which there is an End of Session token), the sender shall not immediately assume the session is closed, but shall wait for the receiver to both transmit its own response packet with a close

session control subpacket (which the original sender must ACK) and to ACK the original sender's packet containing a close session control subpacket.

However, once the receiver has received the packet containing the close session control subpacket; responded with a packet containing its own close session control subpacket; and acknowledged the packet with the close session control subpacket, the receiver still cannot assume the session to be closed. The receiver shall wait at least double the timeout before considering the session to be closed. This guarantees that if the ACK packet was lost, the sender will try to retransmit the packet with the close session control subpacket and the receiver will have a chance to retransmit the ACK. See Figure 17

Figure 17 Closing a Session



4 Life Cycle of SPs

4.1 Life Cycle of SPs Overview

Each SP in a TPer is associated with an attribute called the life cycle state. The access control settings on the SP are derived in part from its life cycle state. This section defines the various life cycle states and transitions that an SP may make between them.

Life cycle applies to each individual SP. The life cycle state of the TPer as a whole emerges from life cycle states of individual SPs.

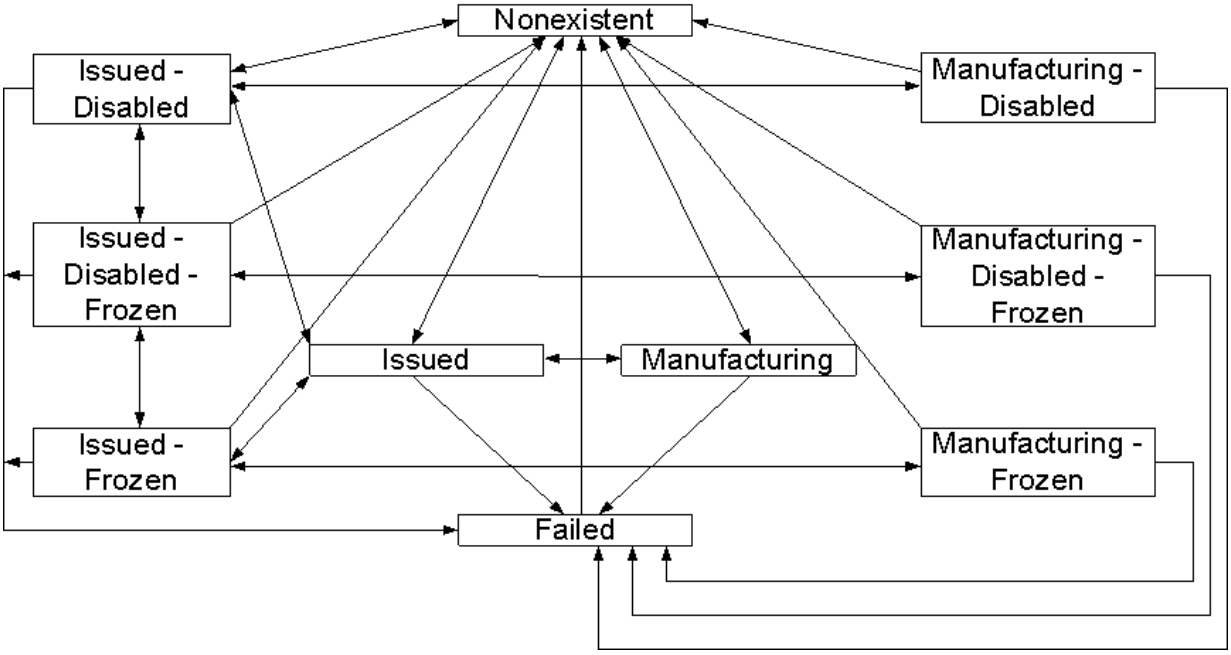
Life cycle in this specification describes the default access control settings of an SP at issuance. The access control settings of an issued SP are out of scope of the specification. An issued SP is operational and may be personalized, and may be participating in life cycle states that are wholly within the control of the SP owner of the issued SP.

Exceptions to this rule, where limitations are placed on personalization or where SP capabilities are frozen, are within the scope of this and any Security Subsystem Class-specific specification.

Template-related exceptions are specified in the related Template-specific life cycle sections of this document.

4.2 Life Cycle States

Figure 18 Life Cycle State Transitions



The following list details the states depicted in Figure 18 and reviews the requirements for each transition between states, including the entry and exit conditions for each state, and the conditions that apply to each of these transitions.

1. **Nonexistent**:
    a. **Definition**: The Nonexistent state is a theoretical state that describes the condition of an SP before it has been instantiated, or after it has been deleted.
    b. **Entry:** This state is “entered” by an SP when that SP is deleted.

c. **Exit:** This state is "exited" when an SP is successfully created, either during the manufacturing process or via successful invocation of the `IssueSP` method from within a session to the Admin SP. An SP may "exit" the Nonexistent state into the following other states:

- **Issued –** successful invocation of the `IssueSP` method causes an SP to be created. The SP shall be created in this state if the SP is operational and if the value of the `IssueSP` method parameter "Enabled" was True.
- **Issued-Disabled** – successful invocation of the `IssueSP` method causes an SP to be created. The SP shall be created in this state if the SP is operational and if the value of the `IssueSP` method parameter "Enabled" was False.
- **Manufacturing** – an SP created during the manufacturing process causes an SP to move from the Nonexistent state to the Manufacturing state.

2. **Issued**:

a. **Definition:** The Issued state is the standard operational state of an SP, and defines the initial required access control settings of an SP based on the Templates incorporated into the SP, prior to personalization. SPs created in Manufacturing enter the Issued state at an implementation-specific point and at that point in time shall have initial access controls settings as defined in this specification.

b. **Entry:** Initial entry to this state is gated by the access control settings on the `IssueSP` method of the Admin SP. An SP may enter the Issued state from the following states:

- **Nonexistent –** Successful invocation of the `IssueSP` method.
- **Manufacturing –** SPs created during the Manufacturing process shall transition to the Issued state prior to personalization and regular SP operation.
- **Issued-Disabled –** may be transitioned into the Issued state by setting the value of the `Enabled` column of the SP's `SPInfo` table to True.
- **Issued-Frozen –** may be transitioned into the Issued state by setting the value of the `Frozen` column of the Admin SP's `SP` table to False.

c. **Exit:** An SP in the Issued state can exit into the following states:

- **Nonexistent –** The SP may be deleted, and thus enter the Nonexistent state, by successful invocation of the `DeleteSP` method from within a session to the SP, or by successful invocation of the `Delete` method on the SP object from within a session to the Admin SP.
- **Manufacturing –** SPs created during the manufacturing process may be transitioned from the Issued state to the Manufacturing state through implementation-specific means.
- **Issued-Disabled –** an SP may be transitioned into the Issued-Disabled state by setting the value of the `Enabled` column of the SP's `SPInfo` table to False.
- **Issued-Frozen –** an SP may be transitioned into the Issued-Frozen state by setting the value of the `Frozen` column of the Admin SP's `SP` table to True.
- **Failed –** an SP may move into the Failed state if an unrecoverable write error or other failure occurs. The TPer controls entry to this state.

3. **Issued-Disabled**:

a. **Definition**: This state occurs after an SP has been issued, when the value of the `Enabled` column of the SP's `SPInfo` table is False.

b. **Entry:** An SP may enter the Issued-Disabled state from the following states:

- **Nonexistent -** successful invocation of the `IssueSP` method causes an SP to be created. The SP shall be created in this state if the SP is operational and if the value of the `IssueSP` method parameter Enabled was False.
- **Issued -** an SP may be transitioned from the Issued state into the Issued-Disabled state by setting the value of the `Enabled` column of the SP's `SPInfo` table to False.
- **Issued-Disabled-Frozen –** an SP may be transitioned from the Issued-Disabled-Frozen state by setting the value of the `Frozen` column of the Admin SP's `SP` table to False.
- **Manufacturing-Disabled –** an SP that had previously entered the Manufacturing-Disabled state may be capable of returning to the Issued-Disabled state. Control of this transition is implementation-specific.

c. **Exit:** An SP may exit the Issued-Disabled state into the following states:

- **Nonexistent –** The SP may be deleted, and thus enter the Nonexistent state, by successful invocation of the `DeleteSP` method from within a session to the SP, or by successful invocation of the `Delete` method on the SP object from within a session to the Admin SP.
- **Issued –** an SP may be transitioned from the Issued-Disabled state into the Issued state by setting the value of the `Enabled` column of the SP's `SPInfo` table to True.
- **Issued-Disabled-Frozen –** an SP may be transitioned from the Issued-Disabled state into the Issued-Disabled-Frozen state by setting the value of the `Frozen` column of the Admin SP's `SP` table to True.
- **Manufacturing-Disabled –** an SP that was created during the manufacturing process may be capable of entering the Manufacturing-Disabled state from the Issued-Disabled state. Control of this transition is implementation-specific.
- **Failed –** an SP may move into the Failed state if an unrecoverable write error or other failure occurs. The TPer controls entry to this state.

4. **Issued-Frozen**:

a. **Definition**: This state occurs after an SP has been issued, when the value of the `Frozen` column of the Admin SP's `SP` table is True.

b. **Entry:** An SP may enter the Issued-Frozen state from the following states:

- **Issued -** an SP may be transitioned from the Issued state into the Issued-Frozen state by setting the value of the `Frozen` column of the Admin SP's `SP` table to True.
- **Issued-Disabled-Frozen –** an SP may be transitioned from the Issued-Disabled-Frozen state into the Issued-Frozen state by setting the value of the `Enabled` column of the SP's `SPInfo` table to True.
- **Manufacturing-Frozen –** an SP that had previously entered the Manufacturing-Frozen state may be capable of returning to the Issued-Frozen state. Control of this transition is implementation-specific.

c. **Exit:** An SP may exit the Issued-Frozen state into the following states:

- **Nonexistent –** The SP may be deleted, and thus enter the Nonexistent state, by successful invocation of the `DeleteSP` method from within a session to the SP, or by successful invocation of the `Delete` method on the SP object from within a session to the Admin SP.
- **Issued –** an SP may be transitioned from the Issued-Frozen state into the Issued state by setting the value of the `Frozen` column of the Admin SP's `SP` table to False.
- **Issued-Disabled-Frozen –** an SP may be transitioned from the Issued-Frozen state into the Issued-Disabled-Frozen state by setting the value of the `Enabled` column of the SP's `SPInfo` table to False.
- **Manufacturing-Frozen –** an SP that was created during the manufacturing process may be capable of entering the Manufacturing-Frozen state from the Issued-Frozen state. Control of this transition is implementation-specific.
- **Failed –** an SP may move into the Failed state if an unrecoverable write error or other failure occurs. The TPer controls entry to this state.

5. **Issued-Disabled-Frozen**:

a. **Definition**: This state occurs after an SP has been issued, when both the value of the `Frozen` column of the Admin SP's `SP` table is True and the value of the `Enabled` column of the SP's `SPInfo` table is False.

b. **Entry:** An SP may enter the Issued-Disabled-Frozen state from the following states:

- **Issued-Disabled –** an SP may be transitioned from the Issued-Disabled state to the Issued-Disabled-Frozen state by setting the value of the `Frozen` column of the Admin SP's `SP` table to True.
- **Issued-Frozen –** an SP may be transitioned from the Issued-Frozen state to the Issued-Disabled-Frozen state by setting the value of the `Enabled` column of the SP's `SPInfo` table to False.
- **Manufacturing-Disabled-Frozen –** an SP in the Manufacturing-Disabled-Frozen state may be able to enter the Issued-Disabled-Frozen state. Control of this transition is implementation specific.

c. **Exit:** An SP may exit the Issued-Disabled-Frozen state into the following states:

- **Issued-Disabled –** an SP may be transitioned from the Issued-Disabled-Frozen state into the Issued-Disabled state by setting the value of the `Frozen` column of the Admin SP's `SP` table to False.
- **Issued-Frozen –** an SP may be transitioned from the Issued-Disabled-Frozen state into the Issued-Frozen state by setting the value of the `Enabled` column of the SP's `SPInfo` table to True.
- **Manufacturing-Disabled-Frozen –** an SP that was created during the manufacturing process may be capable of entering the Manufacturing-Disabled-Frozen state from the Issued-Disabled-Frozen state. Control of this transition is implementation-specific.
- **Failed –** an SP may move into the Failed state if an unrecoverable write error or other failure occurs. The TPer controls entry to this state.

6. **Manufacturing**:

a. **Definition**: The Manufacturing state is an implementation-specific state used for diagnostics, trouble-shooting, factory-creation of SPs, etc. The purpose of the Manufacturing state is to insure data protection for, for instance, device redeployment. Only SPs that were created in Manufacturing can re-enter the manufacturing state. An SP that is not created in manufacturing, but created using the `IssueSP` method, cannot transition into the Manufacturing state.

b. **Entry:** The mechanism that causes entrance to the Manufacturing state is implementation-specific. Whatever mechanism is used to enter the manufacturing state should be gated by access controls that require both a manufacturing authority and the SID authority to be authenticated. Prior to/upon entrance to the Manufacturing state, SPs (other than the Admin SP) must be securely erased. When the Admin SP enters the Manufacturing state from the Issued state, the value of the SID (the associated `C_PIN` credential's `Password` column value) shall revert to the original value (as printed on the drive). An SP may enter the Manufacturing state from the following states:

- **Nonexistent –** an SP created during the manufacturing process causes an SP to move from the Nonexistent state to the Manufacturing state.
- **Issued -** SPs created in the manufacturing may be transitioned from the Issued state to the Manufacturing state through implementation-specific means.

c. **Exit:** The mechanism that causes exit from the Manufacturing state is implementation specific. An SP may exit from the Manufacturing state into the following states:

- **Nonexistent -** The SP may be deleted, and thus enter the Nonexistent state, by successful invocation of the `DeleteSP` method from within a session to the SP; by successful invocation of the `Delete` method on the SP object from within a session to the Admin SP; or by other implementation-specific means.
- **Issued –** SPs created during the Manufacturing process shall transition to the Issued state prior to personalization and regular SP operation.
- **Failed –** an SP may move into the Failed state if an unrecoverable write error or other failure occurs. The TPer controls entry to this state.

7. **Manufacturing-Disabled**:

a. **Definition**: The Manufacturing-Disabled state is an implementation-specific state used for diagnostics, trouble-shooting, factory-creation of SPs, etc. The purpose of the Manufacturing-Disabled state is to insure data protection for, for instance, device redeployment. Only SPs that were created in manufacturing can enter the Manufacturing-Disabled state. An SP that is not created in manufacturing, but created using the `IssueSP` method, cannot transition into the Manufacturing-Disabled state.

b. **Entry:** The mechanism that causes entrance to the Manufacturing-Disabled state is implementation-specific. Whatever mechanism is used to enter the manufacturing state should be gated by access controls that require both a manufacturing authority and the SID authority to be authenticated. Prior to/upon entrance to the Manufacturing-Disabled state, SPs (other than the Admin SP) must be securely erased. An SP may enter the Manufacturing-Disabled state from the following states:

- **Issued-Disabled –** an SP that was created during the manufacturing process may be capable of entering the Manufacturing-Disabled state from the Issued-Disabled state. Control of this transition is implementation-specific.

c. **Exit:** The mechanism that causes exit from the Manufacturing-Disabled state is implementation specific. An SP may exit from the Manufacturing-Disabled state into the following states:

- **Nonexistent -** The SP may be deleted, and thus enter the Nonexistent state, by successful invocation of the `DeleteSP` method from within a session to the SP; by successful invocation of the `Delete` method on the SP object from within a session to the Admin SP; or by other implementation-specific means
- **Issued-Disabled –** an SP in the Manufacturing-Disabled state may be capable of returning to the Issued-Disabled state. Control of this transition is implementation-specific.
- **Failed –** an SP may move into the Failed state if an unrecoverable write error or other failure occurs. The TPer controls entry to this state.

8. **Manufacturing-Frozen**:

a. **Definition**: The Manufacturing-Frozen state is an implementation-specific state used for diagnostics, trouble-shooting, factory-creation of SPs, etc. The purpose of the Manufacturing-Frozen state is to insure data protection for, for instance, device redeployment. Only SPs that were created in manufacturing can enter the Manufacturing-Frozen state. An SP that is not created in manufacturing, but created using the `IssueSP` method, cannot transition into the Manufacturing-Frozen state.

b. **Entry:** The mechanism that causes entrance to the Manufacturing-Frozen state is implementation-specific. Whatever mechanism is used to enter the manufacturing state should be gated by access controls that require both a manufacturing authority and the SID authority to be authenticated. Prior to/upon entrance to the Manufacturing-Frozen state, SPs (other than the Admin SP) must be securely erased. An SP may enter the Manufacturing-Frozen state from the following states:

- **Issued-Frozen –** an SP that was created during the manufacturing process may be capable of entering the Manufacturing-Frozen state from the Issued-Frozen state. Control of this transition is implementation-specific.

c. **Exit:** The mechanism that causes exit from the Manufacturing-Frozen state is implementation specific. An SP may exit from the Manufacturing-Frozen state into the following states:

- **Nonexistent -** The SP may be deleted, and thus enter the Nonexistent state, by successful invocation of the `DeleteSP` method from within a session to the SP; by successful invocation of the `Delete` method on the SP object from within a session to the Admin SP; or by other implementation-specific means
- **Issued-Frozen –** an SP in the Manufacturing-Frozen state may be capable of returning to the Issued-Frozen state. Control of this transition is implementation-specific.
- **Failed –** an SP may move into the Failed state if an unrecoverable write error or other failure occurs. The TPer controls entry to this state.

9. **Manufacturing-Disabled-Frozen**:

a. **Definition**: The Manufacturing-Disabled-Frozen state is an implementation-specific state used for diagnostics, trouble-shooting, factory-creation of SPs, etc. The purpose of the Manufacturing-Disabled-Frozen state is to insure data protection for, for instance, device redeployment. Only SPs that were created in manufacturing can enter the Manufacturing-Disabled-Frozen state. An SP that is not created in manufacturing, but created using the `IssueSP` method, cannot transition into the Manufacturing-Disabled-Frozen state.

b. **Entry:** The mechanism that causes entrance to the Manufacturing-Disabled-Frozen state is implementation-specific. Whatever mechanism is used to enter the manufacturing state should be gated by access controls that require both a manufacturing authority and the SID authority to be authenticated. Prior to/upon entrance to the Manufacturing-Disabled-Frozen state, SPs (other than the Admin SP) must be securely erased. An SP may enter the Manufacturing-Frozen state from the following states:

- **Issued-Disabled-Frozen –** an SP that was created during the manufacturing process may be capable of entering the Manufacturing-Disabled-Frozen state from the Issued-Disabled-Frozen state. Control of this transition is implementation-specific.

c. **Exit:** The mechanism that causes exit from the Manufacturing-Disabled-Frozen state is implementation specific. An SP may exit from the Manufacturing-Disabled-Frozen state into the following states:

- **Nonexistent -** The SP may be deleted, and thus enter the Nonexistent state, by successful invocation of the `DeleteSP` method from within a session to the SP; by successful invocation of the `Delete` method on the SP object from within a session to the Admin SP; or by other implementation-specific means
- **Issued-Disabled-Frozen –** an SP in the Manufacturing-Disabled-Frozen state may be capable of returning to the Issued-Disabled-Frozen state. Control of this transition is implementation-specific.
- **Failed –** an SP may move into the Failed state if an unrecoverable write error or other failure occurs. The TPer controls entry to this state.

10. **Failed**:

a. **Definition**: The Failed state describes the condition where the SP has experienced an unrecoverable write failure; physical read error for the hidden (SP) space; or other unrecoverable failure that prevents access to TCG related functionality and data structures (i.e. the SP is unable to accept method invocations).

b. **Entry**: Entry to this state and access to the SP in this state is controlled by an unrecoverable write failure or other unrecoverable failure. The TPer controls entry to this state. An SP may enter the failed state from any other existent state.

c. **Exit**: The Failed state is a terminal state. The only exit available from the Failed state is to the theoretical Nonexistent state.

Life cycle states are recorded in the `LifeCycleState` column of the Admin SP's `SP` table. This column identifies the state in which the SP currently is. The value of this column shall be changed by the TPer whenever an SP's life cycle state changes.

Access control on reading the SPs available in a TPer, and the life cycle states of those SPs, shall be readable by the Anybody authority on the Admin SP.

4.3 Defined Authorities

The initial authorities that can affect the life cycle states are defined for:

1. Base Template (Table 73) – the Admins Authority (SP owner) and Makers Authority.
2. Admin Template (Table 124) – In addition to the Base Template Authorities, the Issuing (and related) authorities, and the SID (TPer Owner) authority.

These are the only Authorities that are within the scope of the specification. Additional authorities may be defined during SP personalization and operational use, as required and permitted by the access control settings defined here.

4.4 State Behaviors

4.4.1 Access Control

Access rights to method invocations on tables are a function of life cycle. Capabilities enabled by access control may change with transitions between life cycle states. The class authority mechanism is the most basic mechanism associated with changes in life cycle state.

Initial required access control settings for each Template are found in the related Template reference sections.

Life cycle state changes related to access control include:

1. Default access controls associated with the creation of a new table or new rows of a table.
2. Changing settings between life cycle states. The change of the life cycle assertion in the SP Table (in the Admin SP) is a case of changing a setting when changing a life cycle state.
3. Changing readability and write-ability conditions.

4.4.2 Issued

Behavior of an SP in the Issued state is described in the Template Reference sections, and specifically in the sections of the Templates of which the SP has been constructed. Access control settings in those sections apply at the point when an SP has been Issued and before personalization occurs.

4.4.3 Issued-Disabled

If the Log template has been issued into the SP, logging in the SP's default log table may reflect at least the successful use of the disabling and enabling functions, any failed session attempts, and failed attempts to invoke the `DeleteSP` method, dependant on personalization.

Template-specific information related to disabling of an SP that includes that Template is found in the Template's reference section in this document.

In the Issued-Disabled state, only a host application that is able to authenticate to the necessary access controls shall have the ability to re-enable the SP. Only method invocations related directly to re-enabling the SP are successful (access control requirements shall still be fulfilled).

Only the following method invocations to the disabled SP will function (fulfilling appropriate access control requirements shall be required):

- `Authenticate`
- `Set` on the `Enabled` column of the `SPInfo` table. Access control requirements must be met as normal. That can be accomplished either during session startup or using the `Authenticate` method.
- `DeleteSP` – Access control requirements shall be met as normal. That can be accomplished either during session startup or using the `Authenticate` method.

In addition, the disabled state does not affect Session Manager protocol layer methods, and session startup methods shall operate as normal.

The TPer owner or an authorized authority shall still have the ability to invoke the `Delete` method within a session to the Admin SP in order to delete the disabled SP.

4.4.4 Issued-Frozen

If the Log template has been issued into the SP, logging in the SP's default log table may reflect at least the any failed session attempts, authentications, or attempts to invoke the `DeleteSP` method, dependant on personalization.

Only the following method invocations to the SP will function (fulfilling appropriate access control requirements shall be required):

- `Authenticate`
- `DeleteSP` – Access control requirements shall be met as normal. That can be accomplished either during session startup or using the `Authenticate` method.

In addition, the Issued-Frozen state does not affect the Session Manager protocol layer methods, and session startup methods shall operate as normal.

4.4.5 Issued-Disabled-Frozen

If the Log template has been issued into the SP, logging in the SP's default log table may reflect at least the successful use of the disabling and enabling functions, any failed session attempts, and failed attempts to invoke the DeleteSP method, dependant on personalization.

Only the following method invocations to the disabled-frozen SP will function (fulfilling appropriate access control requirements shall be required):

- `Authenticate`
- `Set` on the `Enabled` column of the `SPInfo` table. Access control requirements must be met as normal. That can be accomplished either during session startup or using the `Authenticate` method.
- `DeleteSP` – Access control requirements shall be met as normal. That can be accomplished either during session startup or using the `Authenticate` method.

In addition, the Issued-Disabled-Frozen state does not affect Session Manager protocol layer methods, and session startup methods shall operate as normal.

4.4.6 Manufacturing

Behavior of an SP in the Manufacturing state is implementation-specific.

4.4.7 Manufacturing-Disabled

Behavior of an SP in the Manufacturing-Disabled state is implementation-specific.

4.4.8 Manufacturing-Frozen

Behavior of an SP in the Manufacturing-Frozen state is implementation-specific.

4.4.9 Manufacturing-Disabled-Frozen

Behavior of an SP in the Manufacturing-Disabled-Frozen state is implementation-specific.

4.4.10 Failed

When an SP is in the Failed state, session startup methods to the SP shall respond with an error status and session startup shall not be able to complete.

The TPer owner or an authorized authority may invoke the `Delete` method within a session to the Admin SP in order to delete the failed SP.

4.4.11 Miscellaneous

For life cycle requirements of the cryptographic module that supports the Crypto Template and other TPer cryptographic capabilities, see FIPS 140-2. Requirements for operation of the cryptographic module as cited in that document affect only the cryptographic functionality provided by the TPer – not the data stored in the SPs themselves – though cryptographic module failures may affect the TPer's authentication, session startup, and secure messaging capabilities.

5 SP Reference

5.1 SP Globals

The following sections define variables, functions, constants, or any system attribute that applies to all SPs.

5.1.1 Variable Types Overview

This section provides the definitions of the types used in the rest of this document.

The following are the primitive data types (Base_Types) defined by the specification. How these primitive values are stored in a table cell is implementation dependent. Additional information on these types can be found in 3.2.2.

- **integer**. Signed integer. To differentiate among the type sizes, a size identifier is specified with the type, i.e., a one-byte integer is denoted as `integer_1, etc`.
- **uinteger**. Unsigned integer. To differentiate among the type sizes, a size identifier is specified with the type, i.e. a one-byte integer is denoted as `uinteger_1,` etc.
- **bytes**. A fixed size sequence of bytes that can be used to represent any type of data such as strings, blobs, bit vectors, time/dates, etc.
- **bytes{max=n}**. A variable size sequence of bytes. Invocation of the `Get` method on a table cell with this type of value shall return the exact sequence of bytes as was originally set.

Types are specified using the following format (BNF specification):

```
Type := Base_Type  | Simple_Type | Enumeration_Type | Alternative_Type | List_Type
| Restricted_Reference_Type |  General_Reference_Type | Name_Value_Type |
Struct_Type | Set_Type

Base_Type                  := 0
Simple_Type                := 1 uidref{Type} size
Enumeration_Type           := 2 uinteger uinteger
Alternative_Type           := 3 uinteger uidref{Type}*
List_Type                  := 4 uinteger uidref{Type}
Restricted_Reference_Type  := 5|6 uidref{Table}
General_Reference_Type     := 7|8|9
Name_Value_Type            := 10 name value
Struct_Type                := 11 uinteger uidref{Type}*
Set_Type                   := 12 uinteger uinteger
```

- **Base Type.** The `Base_Type` format describes the pre-installed types. All other types are created using the Base Types as building blocks. The Base Types, except for Null, shall not be used directly. Base Types shall always have a `Size` column value of 0 in the `Type` table.
- **Simple Type.** The `Simple_Type` format defines an instance of one of the `Base_Type` types. The `Simple_Type` always includes a size in the format column, which defines the size for that instance of that `Simple_Type`.
- **Enumeration**. An unsigned integer in a specific range. The `Enumeration_Type` format defines the range of the enumeration, where the first integer specified in the format description is the start value of the enumeration, and the second integer specified is the end value. For example, a range of 0 to 2 inclusive, in plaintext:
    - Pseudo-code example: `enum{0..2}`
- **Alternative**. A value that may be an element of one of the specified types. The `Alternative_Type` format defines a union with the uinteger specifying the number of member types and followed by that many uidref{TypeObjectUID} references to the member types. The type of the value is stored with the value in the cell. For example:

  - Pseudo-code example: `typeOr{boolean,uinteger_4,bytes{7}}`

- **List**. A sequence of values of the same type. The maximum number of elements is specified, and the actual number of elements in the list is stored in the cell. The `List_Type` defines with the first integer the maximal number of elements, while the second uidref{Type} specifies the type of the elements. The elements of the list are not required to be provided in any specified order. However, the elements of the list shall be stored in a cell and returned to the host (with the `Get` method, for example) in the order in which they were received by the TPer.

  - Pseudo-code example: `list[10]{boolean}`

  is a list of `boolean` values, with a maximum of 10 elements.

- **Restricted Reference**. A reference to a row of a specific table. The `Restricted_Reference_Type` defines to which table (uidref{TableUID}) the reference values refer. The reference is to a physical row number (5) or a UID (6) within the table. In this example, TNAME is the name of the referenced table:

  - Pseudo-code example: `uidref{TNAME}`
  - Pseudo-code example: `ref{TNAME}`

  The value of a `ref` is the `uinteger` row number for an array or byte table. The value of a `uidref` is a UID from the `UID` column of a non-byte table. A ref value of 0 or uidref value of 00s serves as a “null pointer”.

- **General Reference**. A reference to a row of some table, to the UID of some object, or to the UID of some table. The `General_Reference_Type` format defines a physical row number (7), a uid of some object (8), or a uid of some table (9). The UID reserved to represent “this SP” is encompassed by a `General_Reference_Type` of 8.

  - Pseudo-code example: `uidref{*}`
  - Pseudo-code example: `ref{*}`

- **Name-Value.** This is a name-value pair. The `Name-Value_Type` format indicator is followed by the name and then the value for the type, where name is a uidref to the `name` type and value is a uidref to the required type of the value.

- **Struct.** This is a combination of different types. The `Struct_Type` format indicator is followed by the number of elements and then uidrefs to the rows in the `Type` table that represent each of those elements. Name-value pairs in structs represent optional components. These may be excluded when passing that struct as a method parameter. When used as a column type, the size must account for inclusion of all of a struct's components.

- **Set**. An unsigned integer in a specific range. The `Set_Type` format defines the range of the enumeration, where the first integer is the start value of the enumeration, while the second integer is the end value. The type itself is not limited to only a single selection from among the choices defined, as in the `Enumeration_Type`. The `Set_Type` provides the host the ability to select more than one of the options. Each shall appear only once in the Set. The Set may hold any amount of selections, from zero to the number of selections.

  - Pseudo-code example: Set{0..2} – Valid values for this set are made up of the following = {}, {0}, {1}, {2}, {0,1}, {0,2}, {1,2}, {0,1,2}.

The `Type` table entries that shall represent the built-in types, and additional system types that are predefined entries of the `Type` table are all types specified in Table 30.

5.1.2 Variable Types

Table 30 describes all of the default column types described in the Template Reference sections of the Core Specification. The `UID`, `Name`, `Format`, `Size`, and `Default` columns identify the column values of the `Type` table. These values shall comprise the `Type` table for every SP, prior to any personalization. These types shall not be able to be changed or deleted.

The UID column shall be the UID for the associated type. The value in the Size column, which represents the width of the column required to store a value of that type, is implementation dependent, as the overhead required to differentiate or identify the type or size of the type stored is not defined in this document, except for base types, whose Size column value shall be 0. For instance, in the case of a max_bytes value, it is necessary to store with the max_bytes value the size of that value. This means that a max_bytes type with a size of 18 bytes requires a column width of 19 or more bytes.

In the case of List types and Struct types, ordering is not important, but the TPer shall return the data to a Get method in the same order in which it had been received or set initially.

The Default column defines the default value for the associated type. Rows with no Default column value in the descriptive table are types that shall have a value specified whenever a column of that type is used in a CreateRow (or other similar) method invocation, or that method invocation shall fail. Other rows, those with values, shall have a uidref in the Type table to a byte table that stores the default value for that type (without the “”). See the Type table description in 5.3.2.5 for information on default column values.

The Description column in the table below is informative only, and is not intended to be part of the Type table implementation.

Note: * in the table below indicates SSC-dependent or implantation-dependent values.

Table 30 Default Type Table Values

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 00 01	NULL	0	0		Base installed type, used to represent a null value. The null value for a particular column is dependent on that column's type. In order to define a legal Null value for a particular type, it is necessary to construct an alternative type where Null is one of the options.
00 00 00 05 00 00 00 02	bytes	0	0		Base installed type, used to represent a value made up of a fixed-size sequence of bytes.
00 00 00 05 00 00 00 03	max_bytes	0	0		Base installed type, used to represent a bytes value that is equal to or less than the size specified for the type instance.
00 00 00 05 00 00 00 04	integer	0	0		Base installed type, used to represent a signed integer.
00 00 00 05 00 00 00 05	uinteger	0	0		Base installed type, used to represent an unsigned integer.
00 00 00 05 00 00 02 01	bytes_12	1 0000000500000002 12			
00 00 00 05 00 00 02 02	bytes_16	1 0000000500000002 16			
00 00 00 05 00 00 02 03	bytes_20_def_00	1 0000000500000002 20		"00s"	
00 00 00 05 00 00 02 04	bytes_32_def_00	1 0000000500000002 32		"00s"	

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 02 05	bytes_32	1 0000000500000002 32			This bytes type is used for, among other things, the Key column of the C_HMAC_256 table.
00 00 00 05 00 00 02 06	version_bytes_4	1 0000000500000002 4		"00 00 00 01"	
00 00 00 05 00 00 02 07	bytes_48_def_00	1 0000000500000002 48		"00s"	
00 00 00 05 00 00 02 08	bytes_64_def_00	1 0000000500000002 64		"00s"	
00 00 00 05 00 00 02 09	uid	1 0000000500000002 8			Used for UIDs
00 00 00 05 00 00 02 0A	certificate	1 0000000500000003 *			Max bytes type used to represent a certificate. The limit on the size of this type is SSC-specific.
00 00 00 05 00 00 02 0B	name	1 0000000500000003 32			Name that generically describes bytes{max=32}, which is used for name columns and method parameters. This type is also used in the Name_Value_Type format.
00 00 00 05 00 00 02 0C	password	1 0000000500000003 32			Max {bytes = 32}, used for PINs
00 00 00 05 00 00 02 0D	max_bytes_32	1 0000000500000003 32			
00 00 00 05 00 00 02 0E	max_bytes_64	1 0000000500000003 64			Generic Max Bytes type, used for logging.
00 00 00 05 00 00 02 0F	int_1_def_0	1 0000000500000004 1		"0"	integer_1 with default of 0
00 00 00 05 00 00 02 10	integer_1	1 0000000500000004 1			
00 00 00 05 00 00 02 11	uinteger_1	1 0000000500000005 1			
00 00 00 05 00 00 02 12	uinteger_128	1 0000000500000005 128			
00 00 00 05 00 00 02 13	uinteger_16	1 0000000500000005 16			
00 00 00 05 00 00 02 14	feedback_size	1 0000000500000005 2			Feedback sizes for AES used in CFB or OFB mode. If AES Mode is CFB, this shall be between 1 and the block length. If AES Mode

ID	Name	Format	Size	Default	Description
					is OFB, this shall be the block size.
00 00 00 05 00 00 02 15	uinteger_2	1 0000000500000005 2			
00 00 00 05 00 00 02 16	uinteger_20	1 0000000500000005 20			
00 00 00 05 00 00 02 17	uinteger_21	1 0000000500000005 21			
00 00 00 05 00 00 02 18	uinteger_24	1 0000000500000005 24			
00 00 00 05 00 00 02 19	uinteger_256	1 0000000500000005 256			
00 00 00 05 00 00 02 1A	uinteger_28	1 0000000500000005 28			
00 00 00 05 00 00 02 1B	uinteger_30	1 0000000500000005 30			
00 00 00 05 00 00 02 1C	challenge_bytes	1 0000000500000005 *			This max bytes type is used to represent a random number in a challenge/response protocol. The max number of bytes in this type are SSC/implementation-dependent, and are based on the cryptographic and communications capability of the TPer.
00 00 00 05 00 00 02 1D	uinteger_32	1 0000000500000005 32			
00 00 00 05 00 00 02 1E	max_bytes_get	1 0000000500000003 *			This is the max bytes type used in the get method to represent data retrieved from a byte table. The actual number of bytes that can be retrieved with a single Get invocation is SSC/implementation dependent, but shall be less than or equal to 4294967295 (the maximum number of bytes that may be stored in a byte table)
00 00 00 05 00 00 02 1F	uinteger_36	1 0000000500000005 36			

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 02 20	uinteger_4	1 0000000500000005 4			
00 00 00 05 00 00 02 21	uint_4_def_0	1 0000000500000005 4		"0"	Uinteger_4 with default of 0
00 00 00 05 00 00 02 22	max_bytes_set	1 0000000500000003 *			This is the max bytes type used in the get method to represent data retrieved from a byte table. The actual number of bytes that can be retrieved with a single Get invocation is SSC/implementation dependent, but shall be less than or equal to 4294967295 (the maximum number of bytes that may be stored in a byte table)
00 00 00 05 00 00 02 23	uinteger_48	1 0000000500000005 48			
00 00 00 05 00 00 02 24	uinteger_64	1 0000000500000005 64			
00 00 00 05 00 00 02 25	uinteger_8	1 0000000500000005 8			
00 00 00 05 00 00 02 26	common_name	1 0000000500000003 32		"Host_Application"	This type is used for the CommonName column. Many tables have values defined for the CommonName column for rows created at issuance. This type defines the default value of rows for user-defined objects.
00 00 00 05 00 00 02 27	uinteger_66	1 0000000500000005 66			
00 00 00 05 00 00 02 28	signed_hash	1 0000000500000003 *			This max_bytes type is used to represent a signed hash. The size limit of this type is based on the TPer's cryptographic communications capabilities, but shall be at least large enough to accomodate the largest signed hash output the TPer is capable of supporting.

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 02 29	response	1 0000000500000003 *			This max_bytes type is used to represent a response to a cryptographic challenge. The size limit of this type is based on the TPer's cryptographic communications capabilities, but shall be at least large enough to accomodate the largest response output the TPer is capable of generating and receiving.
00 00 00 05 00 00 02 2A	session_key_encrypt	1 0000000500000003 *			This max_bytes type is used to represent a session key to be used to encrypt communications in secure messaging. The size limit of this type is based on the TPer's cryptographic communications capabilities, but shall be at least large enough to accomodate the largest encryption key size the TPer is capable of supporting for secure messaging.
00 00 00 05 00 00 02 2B	session_key_integrity	1 0000000500000003 *			This max_bytes type is used to represent a session key to be used to generate a message authentication code in secure messaging. The size limit of this type is based on the TPer's cryptographic communications capabilities, but shall be at least large enough to accomodate the largest MAC key size the TPer is capable of supporting for secure messaging.
00 00 00 05 00 00 02 2C	proof	1 0000000500000003 64			This max_bytes type is used to represent a proof supplied to the TPer for verification or generated by the TPer through cryptographic signing of a hash.
00 00 00 05 00 00 02 2D	exchange_key	1 0000000500000003 *			This max_bytes type is used to represent the exchange key supplied to the TPer upon invocation of the IssueSP method. The size limit of this type is based on the TPer's cryptographic communications capabilities, but shall be at least large enough to accomodate the largest exchange key the TPer is capable of supporting for secure messaging.

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 02 2E	iv	1 0000000500000003 *			This max_bytes type is used to represent an Initialization Vector (IV) used for cryptographic operations. The size limit of this type is based on the TPer's cryptographic communications capabilities, but shall be at least large enough to accomodate the largest IV required by host-requested on-TPer encryption and decryption operations.
00 00 00 05 00 00 02 2F	encrypt_result	1 0000000500000003 *			This max_bytes type is used to represent the result of a host-requested on-TPer encryption operation. The size limit of this type is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 30	decrypt_result	1 0000000500000003 *			This max_bytes type is used to represent the result of a host-requested on-TPer decryption operation. The size limit of this type is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 31	sign_result	1 0000000500000003 *			This max_bytes type is used to represent the result of a host-requested on-TPer signing operation. The size limit of this type is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 32	hash_result	1 0000000500000003 *			This max_bytes type is used to represent the result of a host-requested on-TPer hash operation. The size limit of this type is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 33	hmac_result	1 0000000500000003 *			This max_bytes type is used to represent the result of a host-requested on-TPer hmac operation. The size limit of this type is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 34	xor_result	1 0000000500000003 *			This max_bytes type is used to represent the result of a host-requested on-TPer XOR operation. The size limit of this type is based on the TPer's cryptographic communications capabilities.

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 02 35	max_bytes_256	1 0000000500000003 256			This max_bytes type is used to represent the return result from invocation of the Random method.
00 00 00 05 00 00 02 36	bytes_20	1 0000000500000002 20			This bytes type is used for the Key column of the C_HMAC_160 table.
00 00 00 05 00 00 02 37	bytes_48	1 0000000500000002 48			This bytes type is used for the Key column of the C_HMAC_384 table.
00 00 00 05 00 00 02 38	bytes_64	1 0000000500000002 64			This bytes type is used for the Key column of the C_HMAC_512 table.
00 00 00 05 00 00 02 39	encrypt_max_bytes_input	1 0000000500000003 *			This max_bytes type is used to represent the data input (across the interface) to the Encrypt method. The size limit of this type (which represents the maximum amount of data the TPer is able to receive for each Encrypt invocation) may be SSC-dependent and is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 3A	decrypt_max_bytes_input	1 0000000500000003 *			This max_bytes type is used to represent the data input (across the interface) to the Decrypt method. The size limit of this type (which represents the maximum amount of data the TPer is able to receive for each Decrypt invocation) may be SSC-dependent and is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 3B	sign_max_bytes_input	1 0000000500000003 *			This max_bytes type is used to represent the data input (across the interface) to the Sign method. The size limit of this type (which represents the maximum amount of data the TPer is able to receive for each Sign invocation) may be SSC-dependent and is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 3C	verify_max_bytes_input	1 0000000500000003 *			This max_bytes type is used to represent the data input (across the interface) to the Verify method, and is the data to be verified. The size limit of this type (which represents the maximum amount of data the TPer is able to receive for each Verify invocation) may be SSC-dependent and is based on

ID	Name	Format	Size	Default	Description
					the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 3D	verify_max_bytes_proof	1 0000000500000003 *			This max_bytes type is used to represent the proof input (across the interface) to the Verify method, and is the data to be verified against. The size limit of this type (which represents the maximum amount of data the TPer is able to receive for each Verify invocation) may be SSC-dependent and is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 3E	hash_max_bytes_input	1 0000000500000003 *			This max_bytes type is used to represent the data input (across the interface) to the HashCalc method. The size limit of this type (which represents the maximum amount of data the TPer is able to receive for each HashCalc invocation) may be SSC-dependent and is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 3F	hmac_max_bytes_input	1 0000000500000003 *			This max_bytes type is used to represent the data input (across the interface) to the HMACCalc method. The size limit of this type (which represents the maximum amount of data the TPer is able to receive for each HMACCalc invocation) may be SSC-dependent and is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 40	xor_max_bytes_input	1 0000000500000003 *			This max_bytes type is used to represent the data input (across the interface) to the XOR method. The size limit of this type (which represents the maximum amount of data the TPer is able to receive for each XOR invocation) may be SSC-dependent and is based on the TPer's cryptographic communications capabilities.
00 00 00 05 00 00 02 41	stir_integer	1 0000000500000004 *			This represents the integer parameter used with the Stir method. The size of the integer may be SSC-dependent, and is based on the TPer's cryptographic capabilities.

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 04 01	boolean	2 0 1			Derived type, used to represent True (1) or False (0).
00 00 00 05 00 00 04 02	boolean_def_false	2 0 1		"0"	
00 00 00 05 00 00 04 03	boolean_def_true	2 0 1		"1"	
00 00 00 05 00 00 04 04	messaging_type	2 0 128			This enumeration describes the options for selecting secure messaging. The options for this value are defined in Table 42. 27-128 are reserved values.
00 00 00 05 00 00 04 05	life_cycle_state	2 0 15			Used to represent the current life cycle state. The valid values are: 0 = issued, 1 = issued-disabled, 2 = issued-frozen, 3 = issued-disabled-frozen, 4 = manufacturing, 5 = manufacturing-disabled, 6 = manufacturing-frozen, 7 = manufacturing-disabled-frozen, 8 = failed, 9-15 = reserved
00 00 00 05 00 00 04 06	padding_type	2 0 15			Defines the type of padding used with RSA encryption. '0' identifies the value as None or Null, '1' identifies the padding as that described in PKCS #1 v 1.5, and '2' identifies the padding as that described in PKCS #1 v 2.1. Values 3-15 are reserved for future use.
00 00 00 05 00 00 04 08	auth_method	2 0 23			This describes the enumeration used to represent authentications methods that may be used to authenticate authorities. The valid entries are: 0 = None 1 = Password, 2 = Exhange, 3 = Sign, 4 = SymK, 5 = HMAC, 6 = TPerSign, 7 = TPerExchange, 8-23 = reserved for future use
00 00 00 05 00 00 04 09	log_kind	2 0 23			Used to represent the predefined log messages used in the default Log table. The valid values are: 0 = available, 1 = methodFail,2 = methodSuccess, 3 = authenticateFail,4 = authenticateSuccess,5 = transactOpen,6 = transactCommit, 7 = transactAbort,8 = sessionEnd, 9 = user, 10 = system, 11-23 =

ID	Name	Format	Size	Default	Description
					reserved
00 00 00 05 00 00 04 0A	symmetric_mode	2 0 23			Defines the mode to be used with this AES credential. The valid values are: 0 = ECB, 1 = CBC, 2 = CFB, 3 = OFB, 4 = GCM, 5 = CTR, 6 = CCM, 23 = MediaEncryption, 7-22 reserved for future use.
00 00 00 05 00 00 04 0B	clock_kind	2 0 3			Defines the type of clock currently active. The valid values are: 0 = Timer, 1 = Low, 2 = High, 3 = LowAndHigh
00 00 00 05 00 00 04 0C	log_select	2 0 3			Identifies the scope of the logging for an access control association or authority. The valid values are: 0 = None,1 = LogSuccess,2 = LogFail,3 = LogAlways
00 00 00 05 00 00 04 0D	hash_protocol	2 0 15			Selects which hash algorithm should be used to create a digital signature. Options are: 0 = none, 1 = SHA 1, 2 = SHA 256, 3 = SHA 384, 4 = SHA 512, 5-15 = reserved
00 00 00 05 00 00 04 0E	boolean_ACE	2 0 7			Used to identify "And" and "Or", where "And" is 0, "Or" is 1, and “Not” is 2, and 3-7 are reserved for future use - used to construct ACE Expression
00 00 00 05 00 00 04 0F	adv_key_mode	2 0 7			This enumeration defines when the NextKey is moved to the ActiveKey. 0 = wait for ADVKey_Req, 1 = auto-advance keys
00 00 00 05 00 00 04 10	keys_avail_conds	2 0 7			This enumeration describes the conditions required to assert KeysAvailable in the Locking Template. 0 = None, 1 = Authentication of the authority with Set access to read/write locked columns for the LBA Range
00 00 00 05 00 00 04 11	last_reenc_stat	2 0 7			This enumeration identifies the last attempted re-encryption step. 0 = success, 1 = Read error, 2 = Write Error, 3 = Verify Error

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 04 12	verify_mode	2 0 7			This enumeration defines the verification operation to perform after a sector has been written with a new encryption key. 0 = no verify, 1 = verify enabled, 2-7 = reserved
00 00 00 05 00 00 04 13	reencrypt_request	2 1 16			This enumeration identifies a host re-encryption request value. See section 5.8.2.2 for values.
00 00 00 05 00 00 04 14	reencrypt_state	2 1 16			This enumeration identifies the present Re-encryption state for an LBA range. 1 = Idle, 2 = Pending, 3 = Active, 4= Completed, 5 = Paused, 6-16 = Reserved
00 00 00 05 00 00 04 15	table_kind	2 1 3			Defines the kind of table. The valid values are: 1 = Object, 2 = Array, 3 = Byte
00 00 00 05 00 00 04 16	package_purpose	2 1 32			This enumeration describes the purpose for package creation. 1 = Issuance, 2 = Key Wrapping, 3 = Backup, 4-32 = reserved
00 00 00 05 00 00 06 01	ACE_expression	3 2 0000000500000C04 000000050000040E			This is an alternative type where the options are either a uidref to an ACE object or one of the boolean_ACE options
00 00 00 05 00 00 06 02	row_selection	3 2 0000000500000F01 0000000500001001			This type is used to provide a selection between a uidref to an object table row or a ref to an array table row
00 00 00 05 00 00 06 03	columns	3 2 0000000500000805 0000000500001603			This type represents the alternative type used to define the columns in a table in the CreateRow method. The first selection is used if the table does not have an indexed column(s). The second selection is used if the table does have an indexed column(s).
00 00 00 05 00 00 06 04	uint_ref	3 2 0000000500000211 0000000500000C02			Alternative type with selections for a uinteger_1 or a uidref to an object in the Type table
00 00 00 05 00 00 06 05	row	3 20 0000000500001402 0000000500001403			Used to provide a mechanism to select between a name-value pair where the value is the uidref of an object and a name-value pair where the value is the ref of a table row.
00 00 00 05 00 00 06 06	table_object_ref	3 2 0000000500001001 0000000500001201			This type defines a reference to the uid of a table or the uid of some object.

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 06 07	createrow_result	3 2 0000000500000808 0000000500000809			This alternative type offers 2 options – either a list of ref/uidref pairs (to represent the result of invoking CreateRow on an array table), or a list of uidrefs (to represent the result of invoking CreateRow on an object table).
00 00 00 05 00 00 06 08	next_result	3 2 000000050000080A 000000050000080B			This alternative type offers 2 options – either a list of ref/uidref pairs (to represent the result of invoking Next on an array table), or a list of uidrefs (to represent the result of invoking Next on an object table).
00 00 00 05 00 00 06 09	get_result	3 2 000000050000021E 000000050000080E			This alternative type offers 2 options – either a max bytes type that is to be used to represent the retrieved data from a byte table, or a list of lists of column name/value pairs that is the retrieved data from a non-byte table.
00 00 00 05 00 00 06 0A	set_values	3 2 0000000500000222 0000000500000810			This alternative type offers 2 options – either a max bytes type that is used for data to be set to a byte table, or a list of lists of column name/value pairs, where the name is the name of a column in the non-byte table and the value is the value to be stored in that column.
00 00 00 05 00 00 06 0B	encrypt_input	3 2 0000000500000239 0000000500001607			This alternative type, used for data input to the Encrypt method, offers 2 options – either a max bytes type that is used to transmit across the interface as a method parameter the data to be encrypted, or a cell_block that identifies the location of the data to be encrypted, where this data exists in a table within the SP.
00 00 00 05 00 00 06 0C	decrypt_input	3 2 000000050000023A 0000000500001607			This alternative type, used for data input to the Decrypt method, offers 2 options – either a max bytes type that is used to transmit across the interface as a method parameter the data to be decrypted, or a cell_block that identifies the location of the data to be decrypted, where this data exists in a table within the SP.
00 00 00 05 00 00 06 0D	sign_input	3 2 000000050000023B 0000000500001607			This alternative type, used for data input to the Sign method, offers 2 options – either a max bytes type that is used to transmit across the

ID	Name	Format	Size	Default	Description
					interface as a method parameter the data to be signed, or a cell_block that identifies the location of the data to be signed, where this data exists in a table within the SP.
00 00 00 05 00 00 06 0E	verify_input	3 2 000000050000023C 0000000500001607			This alternative type, used for data input to the Verify method of the data to be verified, offers 2 options – either a max bytes type that is used to transmit across the interface as a method parameter the data to be verified, or a cell_block that identifies the location of the data to be verified, where this data exists in a table within the SP.
00 00 00 05 00 00 06 0F	verify_proof	3 2 000000050000023D 0000000500001607			This alternative type, used for data input to the Verify method of the data to be verified against, offers 2 options – either a max bytes type that is used to transmit across the interface as a method parameter the data to be verified against, or a cell_block that identifies the location of the data to be verified against, where this data exists in a table within the SP.
00 00 00 05 00 00 06 10	hash_input	3 2 000000050000023E 0000000500001607			This alternative type, used for data input to the HashCalc method, offers 2 options – either a max bytes type that is used to transmit across the interface as a method parameter the data to be hashed, or a cell_block that identifies the location of the data to be hashed, where this data exists in a table within the SP.
00 00 00 05 00 00 06 11	hmac_input	3 2 000000050000023F 0000000500001607			This alternative type, used for data input to the HMACCalc method, offers 2 options – either a max bytes type that is used to transmit across the interface as a method parameter the data to be HMACed, or a cell_block that identifies the location of the data to be HMACed, where this data exists in a table within the SP.
00 00 00 05 00 00 06 12	xor_input	3 2 0000000500000240 0000000500001607			This alternative type, used for data input to the XOR method, offers 2 options – either a max bytes type that is used to transmit across the interface as a method parameter the data to be XORed, or a

ID	Name	Format	Size	Default	Description
					cell_block that identifies the location of the data to be XORed, where this data exists in a table within the SP.
00 00 00 05 00 00 06 13	stir_input	3 2 0000000500000241 0000000500000401			This alternative type, used for the input to the Stir method, offers 2 options – either an integer type that is used to seed the Random method upon its next invocation, or a boolean that, if True, indicates that the TPer should seed the Random method internally.
00 00 00 05 00 00 06 14	challenge	3 2 000000050000021C 000000050000020C			This alternative type is used to represent a challenge supplied by one communicator to another, and encompasses both nonces for verification and passwords. This type is made up of a max bytes type that represents a challenge, and a max bytes type that is a password.
00 00 00 05 00 00 08 01	AC_element	4 * 0000000500000601			An AC_Element is a list of ACE_Expressions forming a postfix Authority expression. For example: [32 24 0 8273 1 7728 0] is the list representing the infix ACE Expression:((32 AND 24) OR 8273) AND 7728
00 00 00 05 00 00 08 02	ACL	4 * 0000000500000801			An ACL is represented as a list of uidrefs to ACE objects. The length of the list is SSC-dependant.
00 00 00 05 00 00 08 03	type_ref_list	4 * 0000000500000C02			A list of an SSC-dependent number of uidrefs to objects in the Type table
00 00 00 05 00 00 08 04	row_data	4 * 0000000500001405			Used to provide row data when creating a new row. This is a list of SSC-defined length of name-value pairs.
00 00 00 05 00 00 08 05	columns_list	4 * 0000000500001601			This type defines a list of the column type. The number of elements in the list is SSC-specific
00 00 00 05 00 00 08 06	uint_ref_list	4 2 0000000500000604			List of the Alternative type that contains selections for a uinteger_1 or a uidref to an object in the Type table
00 00 00 05 00 00 08 07	template_list	4 * 0000000500000C08			This type defines a list of uidrefs to objects that appear in the Admin SP's Template table. The number of items in this list is SSC-specific.
00 00 00 05 00 00 08 08	ref_uidref_createrow_list	4 * 0000000500001609			This type defines a list of the struct type that is used to represent a ref-uidref pair, and is for use with the CreateRow method. The

ID	Name	Format	Size	Default	Description
					number of items in this list is SSC/implementation-specific.
00 00 00 05 00 00 08 09	uidref_createrow_list	4 * 0000000500001001			This type defines a list of uidrefs, for use with the CreateRow method.
00 00 00 05 00 00 08 0A	ref_uidref_next_list	4 * 0000000500001609			This type defines a list of the struct type that is used to represent a ref-uidref pair, and is for use with the Next method. The number of items in this list is SSC/implementation-specific.
00 00 00 05 00 00 08 0B	uidref_next_list	4 * 0000000500001001			This type defines a list of uidrefs, for use with the Next method.
00 00 00 05 00 00 08 0C	Table_ref_rows_list	4 * 000000050000080C			This type defines a list of Table object descriptor uids and uintegers, and is used with the GetFreeSpace method. The number of members of this list is limited by the SSC/implementation, but shall be less than or equal to 4294967295, the total number of tables that are creatable on an SP (the number of actually creatable tablesis also SSC/implementation-dependent).
00 00 00 05 00 00 08 0D	get_column_sub_list	4 * 0000000500001601			This type defines a list of column name/value pairs. The number of of members of this list is limited by the SSC/implementation, but shall be less than or equal to 4294967295, the total number of columns that are creatable in a table (the number of actually creatable columns is also SSC/implementation-dependent).
00 00 00 05 00 00 08 0E	get_column_list	4 * 000000050000080D			This type defines a list of a list of column name/value pairs. This is used in the Get method to return the requested values. Each list contained in this list represents a different row. The number of elements that may be contained in this list shall be less than or equal to 4294967295, which is the total number of rows that are creatable in a table (the number of actually creatable rows is also SSC/implementation-dependent).
00 00 00 05 00 00 08 0F	set_column_sub_list	4 * 0000000500001601			This type defines a list of column name/value pairs. The number of of members of this list is limited by

ID	Name	Format	Size	Default	Description
					the SSC/implementation, but shall be less than or equal to 4294967295, the total number of columns that are creatable in a table (the number of actually creatable columns is also SSC/implementation-dependent).
00 00 00 05 00 00 08 10	set_column_list	4 * 000000050000080F			This type defines a list of a list of column name/value pairs. This is used as a parameter of the Set method. Each list contained in this list represents a different row. The number of elements that may be contained in this list shall be less than or equal to 4294967295, which is the total number of rows that are creatable in a table (the number of actually creatable rows is also SSC/implementation-dependent).
00 00 00 05 00 00 0A 01	column_ref	5 0000000400000000			Reference to a row number that must exist in the Column table
00 00 00 05 00 00 0C 01	SPTemplates_ref	6 0000000300000000			
00 00 00 05 00 00 0C 02	Type_ref	6 0000000500000000			Reference to a uid that must exist in the Type table.
00 00 00 05 00 00 0C 03	MethodID_ref	6 0000000500000000			Reference to a uid that must exist in the MethodID table
00 00 00 05 00 00 0C 04	ACE_table_ref	6 0000000800000000			This is a Restricted_Reference_Type, which indicates that the uidref used in this type must be to a uid contained in the ACE table.
00 00 00 05 00 00 0C 05	Authority_ref	6 0000000900000000			Reference to a uid that must exist in the Authority table
00 00 00 05 00 00 0C 06	Certificates_ref	6 0000000A00000000			Reference to a uid that must exist in the Certificates table
00 00 00 05 00 00 0C 07	SP_ref	6 0000020500000000			Reference to a uid that must exist in the Admin SP's SP table
00 00 00 05 00 00 0C 08	Template_ref	6 0000020400000000			Reference to a uid that must exist in the Admin SP's Template table.
00 00 00 05 00 00 0C 09	Table_ref	6 0000000100000000			Reference to a uid that must exist in the SP's Table table.
00 00 00 05 00 00 0F 01	row_ref	7			
00 00 00 05 00 00 0F 02	log_row_ref	7			This is a reference type that shall be used specifically for rows in Log tables. When performing type checking, as part of that type checking the TPer shall validate

ID	Name	Format	Size	Default	Description
					that this is a ref to a row in a Log table.
00 00 00 05 00 00 10 01	row_uidref	8			
00 00 00 05 00 00 10 02	cred_object_uidref	8			This is a reference type that shall be used specifically for uidrefs to credential objects. When performing type checking, as part of that type checking the TPer shall validate that this uidref is to an object in a credential (C_*) table.
00 00 00 05 00 00 12 01	table_ref	9			This type is used to represent a uidref to a Table that is one of the set of all tables in the SP.
00 00 00 05 00 00 12 02	ref_def_00	9		"00 00 00 00 00 00 00 00"	This table ref is to a byte table that defines the default value for this column
00 00 00 05 00 00 12 03	byte_table_ref	9			This is a reference type that shall be used specifically for uidrefs to byte tables. When performing type checking, as part of that type checking the TPer shall validate that this uidref is to a table that is a byte table.
00 00 00 05 00 00 14 01	column-name	10 000000050000020B 000000050000020B			Name-value pair that takes a name as the value. The "Name" in the "Name-value" pair shall be "Name", so that use of this type shall be "Name=<name>".
00 00 00 05 00 00 14 02	row_uidref-name	10 000000050000020B 0000000500001001			Used to identify a name-value pair where the value is the uidref of an object. The name to be used with this type is "UID".
00 00 00 05 00 00 14 03	row_ref-name	10 000000050000020B 0000000500000F01			Used to idenfiy a name-value pair where the value is the ref of a table row. The name to be used with this type is "Ref".
00 00 00 05 00 00 14 04	table_ref-name	10 000000050000020B 0000000500001201			This type is used to represent a name-value pair where the value is a uidref to a Table that is one of the set of all tables in the SP. The name in this type shall be "Name".
00 00 00 05 00 00 14 05	type_ref-name	10 000000050000020B 0000000500000C02			Name-value pair that takes as a value a reference to a uid that must exist in the Type table. The Name in this Name-value pair shall always be "Type".
00 00 00 05 00 00 14 06	name-uinteger_2	10 000000050000020B 0000000500000215			Name-value pair that takes a uinteger_ as the value.

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 14 07	name-uinteger_1	10 000000050000020B 0000000500000211			Name-value pair that takes a uinteger_1 as the value.
00 00 00 05 00 00 14 08	name-startColumn	10 000000050000020B 000000050000020B			Name-value pair used for the cell_block type. The name portion of this type shall be "startColumn".
00 00 00 05 00 00 14 09	name-endColumn	10 000000050000020B 000000050000020B			Name-value pair used for the cell_block type. The name portion of this type shall be "endColumn".
00 00 00 05 00 00 14 0A	name-startRow	10 000000050000020B 0000000500000605			Name-value pair used for the cell_block type. The name portion of this type shall be "startRow".
00 00 00 05 00 00 14 0B	name-endRow	10 000000050000020B 0000000500000605			Name-value pair used for the cell_block type. The name portion of this type shall be "endRow".
00 00 00 05 00 00 16 01	column	11 2 0000000500001401 0000000500001405			This type defines a column name and its associated type.
00 00 00 05 00 00 16 02	lag	11 2 0000000500001406 0000000500001406			A struct made up of 2 uinteger_2 name-value types, used to define the lag when setting time. The 2 types represent seconds and fraction of seconds. The names required are "Seconds" for the first value and "Fraction" for the second. The "Fraction" value is a number of milliseconds.
00 00 00 05 00 00 16 03	columns_struct	11 2 0000000500000805 0000000500000805			This type is a struct made up of two Columns_list types. The first list is the indexed columns of a table, and the second list is the rest of the columns in the table. This is used in table creation.
00 00 00 05 00 00 16 04	date	11 3 0000000500001406 0000000500001407 0000000500001407			The date type represents the date portion of the time from the system clock. This is a set of name-value pairs, with the following names: "Year" (uinteger_2), "Month" (uinteger_1), and "Day" (uinteger_1) (see 5.5.5.6)

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 16 05	clock_time	11 3 0000000500001406 0000000500001608 0000000500001406			Type made up of name-value pairs used to represent time. Any value not supplied is treated as 0. Time comes from the Clock SP. If the host has supplied a trusted time since powerup, that time is used; otherwise a monotonic counter is used. The Clock_time type can be used to represent times in either Generalized Time or UTC Time. Using this type to represent UTC Time requires 0's (zeroes) in fields where Generalized time requires a value but UTC Time does not (i.e. 2006 in UTC Time would be represented as 0006). The names for these name-value types are "Year", "Month", "Day", "Hour", "Minute", "Second", "Fraction" (see 5.5.5.6)
00 00 00 05 00 00 16 06	type_def	11 3 uinteger_* 0000000500000806 0000000500000803			A struct made up of a Uinteger of SSC-dependent size and a list of uidrefs to objects in the Type table
00 00 00 05 00 00 16 07	cell_block	11 5 0000000500001404 000000050000140A 000000050000140B 0000000500001408 0000000500001409			Struct type that is used to represent a rectangular range of a table. An area between the whole table and a single cell can be selected. NOTE: The parts are optional types. The Table defaults to the table being operated on, if there is one (Get, for example). The rows and columns default to the first or last, as appropriate.For example:[Table="MyTable"] refers to the entire table.[Table="MyTable", startRow=2, endRow=5], refers to all columns of rows 2 through 5, inclusive, of table "MyTable". When referencing an object table row from within a method, startRow and endRow are the ID of the object (and must be the same). The name-values for these are, “Table=uid”, “startRow=uid”, “endRow=uid”, “startColumn=name”, “endColumn=name” (as indicated in the individual types that make up this struct)

ID	Name	Format	Size	Default	Description
00 00 00 05 00 00 16 08	struct-name-uinteger_1	11 5 0000000500001407 0000000500001407 0000000500001407 0000000500001407 0000000500001407			Struct composed of 5 Name-uinteger_1 types, used to construct the clock_time type. The names required are "Month", "Day", "Hour", "Minute", "Second" (see 5.5.5.6)
00 00 00 05 00 00 16 09	struct-ref_uidref	11 2 0000000500000F01 0000000500001001			This is a struct composed of 2 types – a row_ref (value of an Array tables RowNumber column) and a row_uidref (value of a non-Byte table's UID column)
00 00 00 05 00 00 16 0A	struct-Table_ref_uint_4	11 2 0000000500000C09 0000000500000220			This is a struct composed of 2 types – a reference by uid to a Table descriptor object, and a uinteger 4 value.
00 00 00 05 00 00 18 01	reset_types	12 0 31			This Set type is used to identify TCG reset types that map to interface specific behaviors. The set values are: 0 = Power Cycle, 1 = Hardware, 2 = HotPlug, 3-15=reserved for TCG use, 16-31 reserved for vendor-specific reset behaviors.
00 00 00 05 00 00 18 02	gen_status	12 0 63			This set type is used to identify the general status of the re-encryption process. See section 5.8.2.2 for values.
00 00 00 05 00 00 18 03	enc_supported	12 0 15			This set describes the types of user data encryption supported by the TPer. 0 = None, 1 = Media Encryption, 2-15 are reserved.

5.1.3 SP Method Status Codes

SP method calls invoke specific operations and receive associated status. The following sections identify and define the status codes that may be received in response to method invocations and other operations. Table 31 identifies the value associated with each of these status codes.

Table 31 Status Codes

Name	Value
SUCCESS	0
NOT_AUTHORIZED	1
READ_ONLY	2
SP_BUSY	3
SP_FAILED	4
SP_DISABLED	5
SP_FROZEN	6
NO_SESSIONS_AVAILABLE	7

Name	Value
INDEX_CONFLICT	8
INSUFFICIENT_SPACE	9
INSUFFICIENT_ROWS	10
INVALID_COMMAND	11
INVALID_PARAMETER	12
INVALID_REFERENCE	13
INVALID_SECMSG_PROPERTIES	14
TPER_MALFUNCTION	15
TRANSACTION_FAILURE	16
RESPONSE_OVERFLOW	17

5.1.3.1 SUCCESS

This status is returned when a method executes without error.

5.1.3.2 NOT_AUTHORIZED

This response is returned whenever an attempt is made to invoke a method for which the host does not have authorization.

5.1.3.3 READ_ONLY

This response is returned if a method that requires invocation from within a Read-Write session is invoked from within a Read-Only session. `DeleteSP` is an example of a method that may return this status.

5.1.3.4 SP_BUSY

This status shall be returned if an attempt is made to open a Read-Write session to an SP when any other session to that SP is already open, or when an attempt is made to open a Read-Only session to an SP with which a Read-Write session is already open.

5.1.3.5 SP_FAILED

This status may be returned if an attempt is made to open a session to an SP that is in the Failed life cycle state.

5.1.3.6 SP_DISABLED

This status may be returned if a method is invoked from within a session to an SP that is in the Issued-Disabled or Issued-Disabled-Frozen state, and the method is not permitted because of the limitations placed on SP operation by the state behavior.

5.1.3.7 SP_FROZEN

This status may be returned if a method is invoked from within a session to an SP that is in the Issued-Frozen or Issued-Disabled-Frozen state, and the method is not permitted because of the limitations placed on SP operation by the state behavior.

5.1.3.8 NO_SESSIONS_AVAILABLE

This status is returned if an attempt is made to open a session on a TPer on which the maximum number of concurrent sessions available for use are already being used.

5.1.3.9 INDEX_CONFLICT

This occurs when a conflict between objects is created due to the attempt to create a second object with a unique index that is already in use by another object. For instance, this status may be received when attempting to create a table, when a table already exists with the name submitted in the `CreateTable` invocation.

5.1.3.10 INSUFFICIENT_SPACE

This status is returned if an attempt is made to:

- Create an SP and there is insufficient space on the TPer to create the new SP
- Create a table and there is insufficient space in the SP to create the new table
- Create more rows in a table than is permitted by the TPer or by the table's size settings.

Note that it is possible that re-invoking the method and requesting a smaller size for the SP or table may enable the method to then complete properly.

5.1.3.11 INSUFFICIENT_ROWS

This command may be returned if an attempt is made to create a table or object, but the associated metadata or support table rows (i.e., the `Table`, `Column`, `Method`, or `ACE` tables) cannot be created to support the new object or table.

5.1.3.12 INVALID_COMMAND

This status is returned if a method call cannot execute due to attempted invocation of an invalid or nonexistent method.

5.1.3.13 INVALID_PARAMETER

This status is returned if a method invocation has any invalid parameters. There are many situations in which this error could be returned. Some of the specific situations where this could occur are:

- Columns specified in the `CreateRow` method invocation are not part of the table definition.
- If an attempt is made to set a cell to a value larger (or smaller) than that cell's type allows.
- If a specified cell_block parameter value is not a valid cell_block for the method.
- If an incorrect credential type is parameterized.

5.1.3.14 INVALID_REFERENCE

This status is returned if a parameter (of either the invoking, i.e., table or object, parameter or the invoked parameters) is to a table row that includes an invalid or incorrect reference to an object, SP, table, or other table row.

5.1.3.15 INVALID_SECMSG_PROPERTIES

This status is returned if the host attempts to explicitly authenticate an authority for which the session properties are not appropriate, for instance if the host requires secure messaging but secure messaging is not in operation for the session, or if a different secure messaging type than is required by the authority is in operation for the session.

5.1.3.16 TPER_MALFUNCTION

This status is returned when some operational failure has occurred within the TPer that has caused the method invocation to fail.

5.1.3.17 TRANSACTION_FAILURE

This status is returned when a method fails due to a failure of the method due to the transactional context in which it was invoked. An example of this is if a TPer is unable to process within the transaction the amount of data supplied as a parameter of the method, which under other circumstances the TPer would be able to process. The TPer in this case would return this status code to indicate that the method failed due to the transactional context, not due to a problem with the method invocation itself.

5.1.3.18 RESPONSE_OVERFLOW

This status is returned when a method fails if the method response and associated protocol overhead do not fit entirely within the response buffer.

5.2 Session Manager Methods

5.2.1 Overview

Session Manager protocol layer methods permit a host to retrieve information about a TPer without having to start a session, and provide the methods required to enable session startup.

Due to the nature of the Session Manager protocol layer methods, the responses to methods at this protocol layer are formatted as methods from the TPer to the host. In the case of multiple method invocations by a host to a TPer on the Session Manager layer, this mechanism allows the host to identify the method to which a response is directed.

Session Manager methods are always invoked using an InvokingUID of SMUID, which is the reserved UID `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF`. Session Manager methods may be listed in the `MethodID` table for each SP, but this is not required.

5.2.2 TPer Properties Method

5.2.2.1 Properties (Method)

The `Properties` method pertains to the exchange of session-related metadata and settings between the host and the TPer prior to session start-up.

```
SMUID.Properties[ HostProperties = list{name, value} ]
=>
SMUID.Properties[ Properties : list{name, value}, HostProperties = list{name, value}
]
```

This Session Manager layer method is used by the host to provide its communication properties to the TPer and retrieve the communication properties of the TPer.

A list of name/value pairs may be provided as arguments when invoking the `Properties` method.

If the method is successfully invoked with an empty list of arguments, then the response is a list of property names and values from the TPer. All property name/value pairs defined in this specification shall be returned in the response, in addition to any other name/value pairs that the TPer may have. The host application shall ignore any that it does not understand. The TPer shall return values for the names described in Table 32 or in the associated SSC (the values in the SSC have precedence). The values returned shall apply to all sessions started with the currently associated ComID.

The properties to be returned are stored in the `Properties` table in the Admin SP. The format of this table is defined in the Admin Template section of this specification (section 5.4.2.4). All of the property name/value pairs, both specified and implementation-defined, shall be stored in the `Properties` table.

If the method is invoked with a list of parameters, the list of name/value pairs that the TPer shall recognize are **MaxPacketSize, MaxComPacketSize, MaxResponseComPacketSize, MaxIndTokenSize,** and **MaxAggTokenSize** as described in Table 32. These parameters are used to describe the communications capabilities that the host possesses, and apply to any sessions started using the ComID associated with this `Properties` method invocation.

The TPer shall use these host properties when it is constructing responses to be transmitted to the host. The host may omit properties as necessary, depending on the host's communications capabilities. If the host specifies a value for a property that does not meet the minimum requirement as defined in Table 32, then the TPer shall use the minimum value defined in Table 32 in place of the value supplied by the host.

If the host includes property parameters to the `Properties` method invocation, then the TPer's response shall include the communication property value settings it will use during the session (both for its communications and the host's). These settings shall apply to all sessions started with the currently associated ComID.

If a host includes property parameters to the `Properties` method invocation that the TPer does not recognize or comprehend, the TPer shall ignore those parameters, and not return them in its response.

Because of the asynchronous nature of the Session Manager protocol layer and the possible different ordering of responses to Session Manager layer methods, the response to this method is formatted as a `Properties` method invocation so as to be identifiable as the response to the `Properties` method.

Table 32 Properties Method Response

Property	Type	Description
SessionVersion	version	The version number of the TPer firmware.
MaxPacketSize	uinteger_4	The maximum size of a packet (including both data and header), in bytes, that the communicator is able to receive. This value shall be at least 256. A value of 0 indicates no limit.
MaxComPacketSize	uinteger_4	The maximum size of an IF Command payload (includes both the ComPacket header and payload) that the communicator is able to receive. A value of 0 indicates no limit.
MaxResponseComPacketSize	uinteger_4	The maximum length of an IF Command payload that the communicator is able to generate. A value of 0 indicates no limit.
MaxSessions	uinteger_2	The maximum number of simultaneous sessions supported by the TPer. A value of 0 indicates no limit.
MaxReadSessions	uinteger_2	The maximum number of simultaneous Read-Only sessions to any one SP supported by the TPer. A value of 0 indicates no limit.
MaxIndTokenSize	uinteger_4	The maximum size of a token (in bytes) in a single subpacket that the communicator is able to accept. This value shall be at least 256. A value of 0 indicates no limit.
MaxAggTokenSize	uinteger_4	The maximum aggregate size of a token spanning multiple subpackets that the communicator is able to accept. This value shall be at least 256. A value of 0 indicates no limit.
MaxAuthentications	uinteger_2	The maximum number of simultaneously authenticated individual authorities per session that the TPer is able to support. A value of 0 indicates no limit.
MaxTransactionLimit	uinteger_2	The maximum number of concurrently open transactions that the TPer is able to support in a single session. A value of 0 indicates no limit.
DefSessionTimeout	uinteger_8	The session timeout length (in milliseconds) used by the TPer by default. A value of 0 indicates no limit.
MaxSessionTimeout	uinteger_8	The longest supported session timeout length (in milliseconds) supported by the TPer. A value of 0 indicates no limit.
MinSessionTimeout	uinteger_8	The shortest supported session timeout length (in milliseconds) supported by the TPer. A value of 0 indicates no limit.
DefTransTimeout	uinteger_4	The transmission timeout length (in milliseconds) used by the TPer by default. A value of 0 indicates no limit.
MaxTransTimeout	uinteger_4	The longest transmission timeout length (in milliseconds) permitted by the TPer. A value of 0 indicates no limit.

Property	Type	Description
MinTransTimeout	uinteger_4	The shortest transmission timeout length (in milliseconds) permitted by the TPer. A value of 0 indicates no limit.
MaxComIDTime	uinteger_8	The timeout length (in milliseconds) used by the TPer after it has assigned a ComID. A session using the associated ComID shall be started within this interval or the ComID shall transition from Issued to Inactive. . A value of 0 indicates no limit.
MaxComIDCMD	uinteger_4	SSC-dependent limit on the number of interface commands that may be issued using a specific ComID. A value of 0 indicates no limit. . A value of 0 indicates no limit.
RealTimeClock	boolean	Identifies if a real time clock is present on the TPer

5.2.3 Session Startup Methods

5.2.3.1 StartSession/SyncSession Methods

```
SMUID.StartSession[
HostSessionID : uinteger_4,
SPID : SP_ref,
Write : boolean,
HostChallenge = challenge,
HostExchangeAuthority = Authority_ref,
HostExchangeCert = certificate,
HostSigningAuthority = Authority_ref,
HostSigningCert = certificate,
SessionTimeout = uinteger_8,
TransTimeout = uinteger_4,
InitialCredit = uinteger_2,
SignedHash = signed_hash ]
=>
SMUID.SyncSession[
HostSessionID : uinteger_4,
SPSessionID : uinteger_4,
SPChallenge = challenge,
SPExchangeCert = certificate,
SPSigningCert = certificate,
TransTimeout = uinteger_4,
InitialCredit = uinteger_2,
SignedHash = signed_hash ]
```

The **HostSessionID** parameter in the `StartSession` invocation is the host-side session number assigned and used by the host to identify this session. All further invocations in this series of method invocations and responses will use this host-assigned session number in the HostSessionID parameter.

The **SPID** parameter in the `StartSession` invocation is the uid of the SP with which the host is attempting to start a session. This is the uid of the SP in the Admin SP's `SP` table.

The **Write** parameter determines the type of session that is being started. This value is True when a Read-Write session is requested and False when a Read-Only session is requested.

If the Signing Authority (identified in the **HostSigningAuthority** parameter) calls out a `C_PIN` credential, then the **HostChallenge** parameter is used by the host to submit a password for authentication. Otherwise, this parameter is used to submit a nonce to the SP that, during secure session startup, will return a response that will be based on the HostChallenge value and the authentication requirements of the Signing Authority.

The **HostExchangeAuthority** identifies the authority whose credential will be used to exchange keys with the SP.

The optional **HostExchangeCert** parameter provides the certificate associated with the credential to be used with the HostExchangeAuthority.

The **HostSigningAuthority**'s credential is used to formulate a response to the SP's challenge, and is used to sign the method hash.

See section 5.3.4.1.4 for more information on how authorities interact during session startup.

The optional **HostSigningCert** parameter provides attestation to the HostSigningAuthority's credential.

The **SessionTimeout** parameter is used to allow the host to provide a requested timeout value for the session. The value, in milliseconds, shall be less than the TPer's MaxSessionTimeout property and greater than the TPer's MinSessionTimeout property (see Table 32), as well as less than the value of the `SPSessionTimeout` column in the SP's `SPInfo` table. If no value is specified for this parameter, then the SP's default value, stored in the `SPInfo` table's `SPSeessionTimeout` column, shall be used. If no value exists as an SP default (i.e. the `SPSessionTimeout` column value is zero), then the TPer default (as reported in the `Properties` method response, DefSessionTimeout) shall be used.

The **TransTimeout** parameter is used to allow the host to provide a requested timeout value for acknowledgement. The value, in milliseconds, shall be less than the TPer's MaxTransTimeout property and greater than the TPer's MinTransTimeout property (these values are reported as the results of the `Properties` method – see Table 32).

If this capability is supported and no value is specified for this parameter, then the TPer's default value (identified as the DefTransTimeout response to the `Properties` mehod), shall be used as the transmission timeout value. For more information on the transmission timeout mechanism, see 3.4.6.3.

The **InitialCredit** parameter enables the host to provide an amount of credits to the TPer for use in data exchange once the session has been successfully opened. For more information on the buffer management/flow control mechanism, see 3.4.6.4.

The optional **SignedHash** parameter of each session startup method is present if hashing is required by the Control Authority for that communicator (see section 3.4.4.7). This is a signed hash of all the other parameters to the method, other than the SignedHash parameter. The purpose of this MAC is to provide integrity during session startup, prior to the point when secure messaging begins.

The Host Control Authority identifies the hash type and signing type if hashing has been called out on messages from the Host to the SP. The SP Control Authority, if referenced by the Host Control Authority, identifies the hash type and signing type if hashing has been called out on messages from the SP to the host (see section 3.4.4.7).

The **HostSessionID** parameter in the `SyncSession` invocation is the same as that in the `StartSession` invocation.

The **SPSessionID** parameter in the `SyncSession` invocation is the TPer side session number, which is assigned by the TPer.

The **SPChallenge** parameter is required if the `StartSession` invocation includes a HostSigningAuthority that directly invokes a signing credential. Otherwise, this parameter will be omitted.

The **SPExchangeCert** and **SPSigningCert** are the certificates for the credentials referenced by the authorities that may be called out by the HostSigningAuthority specified in the `StartSession` invocation.

The **TransTimeout** parameter in the `SyncSession` method is used by the TPer to report the Timeout value it will use. This optional parameter shall be larger than the value of the TransTimeout parameter of the `StartSession` method. This parameter is used to allow the TPer to provide a transmission timeout value for acknowledgement larger than that requested by the host.

The TransTimeout parameter value (measured in milliseconds) shall be less than the TPer's MaxTransTimeout property and greater than the TPer's MinTransTimeout property (see Table 32).

If this capability is supported and no value is specified for this parameter in either the `StartSession` or `SyncSession` methods, then the TPer's default value (identified as the DefTransTimeout response to the `Properties` mehod), shall be used as the transmission timeout value. For more information on the transmission timeout mechanism, see 3.4.6.3.

The **InitialCredit** parameter enables the TPer to provide an amount of credits to the host for use in data exchange once the session has been successfully opened. For more information on the buffer management/flow control mechanism, see 3.4.6.4.

The **SignedHash** of the `SyncSession` method, if present, is the hash of the method's parameter's signed by the response signing credential that is the credential referred to by the SPSigningAuthority.

5.2.3.2 StartTrustedSession/SyncTrustedSession Methods

```
SMUID.StartTrustedSession[
HostSessionID : uinteger_4,
SPSessionID : uinteger_4,
HostResponse = response,
HostEncryptSessionKey = session_key_encrypt,
HostIntegritySessionKey = session_key_integrity,
SignedHash = signed_hash ]
=>
SMUID.SyncTrustedSession[
HostSessionID : uinteger_4,
SPSessionID : uinteger_4,
SPResponse = response,
SPEncryptSessionKey = session_key_encrypt,
SPIntegritySessionKey = session_key_integrity,
SignedHash = signed_hash ]
```

Note: The `StartTrustedSession`/`SyncTrustedSession` method exchange, if needed, can only occur directly after the `StartSession`/`SyncSession` method exchange. If called any other time, the attempted method invocation shall return an error result.

The **HostResponse** is included if the `SyncSession` method contained an SPChallenge argument. The response is dictated by the credential of the HostSigningAuthority.

The **HostEncryptSessionKey** is the session keyset generated by the host and encrypted with the key used for exchange with the SP (see Session Startup (section 5.3.4.1.4) for more information). This session keyset is used in secure messaging to encrypt packets sent from the host to the SP.

The **HostIntegritySessionKey** is the session keyset generated by the host and encrypted with the key used for exchange with the SP. This session keyset is used to create a MAC of the transmitted data (if required) to aid in integrity assurance.

The **SPResponse** argument is included if the `StartSession` method contained a HostChallenge argument. The response is dictated by the `Operation` column value and credential of the SPSigningAuthority.

The **SPEncryptSessionKey** is the session keyset generated by the SP and encrypted with the key used for exchange with the host (see Session Startup (section 5.3.4.1.4) for more information). This session keyset is used in secure messaging to encrypt packets sent from the SP to the host.

The **SPIntegritySessionKey** is the session keyset generated by the host and encrypted with the key used for exchange with the host. This session keyset is used to create a MAC of the transmitted data (if required) to aid in integrity assurance.

For details on using the session startup methods with Elliptic Curve parameters and EC-DH or EC-MQV, see section 5.3.6.14 and 5.3.6.15 respectively.

5.2.3.3 CloseSession Method

```
SMUID.CloseSession[
RemoteSessionNumber : uinteger_4,
LocalSessionNumber : uinteger_4 ]
```

`CloseSession` is a Session Manager protocol layer method. The parameters are two unsigned integers: the first parameter, **RemoteSessionNumber**, is the session number that was used by the remote/receiving entity, and the second parameter, **LocalSessionNumber**, is the session number that was used by the local/sending entity.

This method shall only be able to be transmitted by the TPer. If the session is currently active, the TPer transmits this method to notify the host that it is aborting the session and all open un-committed transactions. The host shall end a session by including an End of Session token in a tokenized message to the TPer (see section 3.2.3.2.5 and 3.4.4.4).

5.3 Base Template

5.3.1 Overview

The Base Template defines the tables and methods that shall be incorporated into all SPs.

5.3.1.1 Base Template Tables and Methods Overview

Base Template tables can be categorically divided into the following groups:

- General Metadata tables – store an SP's self-descriptive information, such as SP identification, size, and version numbers.
- Table and Method Metadata tables – store data about the tables and methods that make up this SP.
- Access Control tables – define which authorities have access to which table/method, object/method, or SP/method combinations, and the secrets and authentication methods those authorities require.
- Credential tables – define available encryption/decryption algorithms and authentication mechanisms, and also store associated secrets or keys. Rows with a gray background in the credential table description tables describe columns in tables that may be hidden in an implementation-specific manner.

Base Template methods are divided into the following groups:

- Basic Table – enable creation of tables, addition and deletion of rows to tables, and modification of table cell values.
- Access Control – define which authorities may execute which methods, request authentication, and modify ACLs.

5.3.2 Data Structures

5.3.2.1 General Metadata Group - SPInfo (Array Table)

The table in this section describes the data that the SP keeps about itself.

Table 33 SPInfo Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	Unique identifier of this row of the SPInfo table (Read-only)
SPID	uid	Unique identifier of this SP as assigned in the Admin SP's SP table. (Read-only)
Name	name	Name of the SP. This shall be the same as the name recorded for this SP in the Admin SP's SP table. (Read-only)
Size	uinteger_8	Total space allocated for the SP at issuance, in bytes. This value will be the same as the value of the Bytes column in the Admin SP's SP table. (Read-only)
SizeInUse	uinteger_8	In bytes, the amount of the allocated space that is in use (for tables). (Read-only)
SPSessionTimeout	uinteger_4	Length of timeout interval (in milliseconds) that this SP uses. (Read-only)
Enabled	boolean	True if the SP is Enabled, False if SP is Disabled. Initial access control over modification of this column permits only the SP Owner (i.e. the Admins class authority) to disable or reenable this SP. When the value of this column is False, the operation of the SP is modified according to 5.3.5.1.

The `SPInfo` table of each SP contains information about the SP, and a copy of some relevant information from the Admin SP. This table has exactly one row.

The `SPID` of the `SPInfo` table and the `GUDID` of the `TPerInfo` table in the Admin SP form an sp_guid that uniquely identifies the SP.

5.3.2.2 General Metadata Group - SPTemplates (Array Table)

The table in this section describes the data that the SP keeps about itself.

Table 34 SPTemplates Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	The unique identifier of this row of the SPTemplates table (Read-only)
TemplateID	Template_ref	The UID of the template as assigned in the Template table of the Admin SP. (Read-only)
Name	name	Name of the Template used as a component in the creation of this SP – this will be the same as the Name recorded in the Admin SP's Template table for the associated template. (Read-only)
Version	version_bytes_4	Initially the 4-byte values will be `0x00 0x00 0x00 0x01`. The values in this table identiy the format of all other tables. The formats documented here in this specification are all version `0x00 0x00 0x00 0x01` formats. (Read-only)

The `SPTemplates` table is an array table that identifies the component templates used to form the SP. There is one row for each Template used to create the SP, including a row for the Base Template for all SPs, and one for the Admin Template in the Admin SP's `SPTemplates` table.

The value of the TemplateID column is the UID assigned to this template in the Admin SP's Template table.

The value of the Name column is the same as the value of the Name column of the Admin SP's Template table for this template.

The value of the Version column refers to TCG defined versions of templates.

5.3.2.3 Table and Method Metadata Group - Table (Object Table)

The table in this section describes the metadata that the SP keeps about all of its tables.

Table 35 Table Table Description

Column	IsIndex	Type	Description
UID		uid	The UID of this row in the Table table. (Read-only)
Name	Yes	name	The name of the table. (Read-only for pre-personalization tables)
CommonName	Yes	name	A name that may be shared among multiple table descriptor objects. (Read-only for pre-personalization tables)
TemplateID	Yes	SPTemplates_ref	This is this Template's UID in the SPTemplates table. This may be zeroes in the Admin SP. (Read-only)
Kind		table_kind	The table type. (Read-only)
Column		column_ref	This is a reference to the Column table row of this table's first column. For byte tables this value will be 0. (Read-only)
NumColumns		uinteger_4	Number of columns in the table. For byte tables this will be 1. (Read-only)
Rows		uinteger_4	Number of rows allocated for the table. (Read-only)
RowsFree		uinteger_4	Number of free rows in the table. (Read-only)
RowBytes		uinteger_4	Number of bytes in each row of the table. This includes bytes devoted to overhead for system columns, type identification, etc. (Read-only)
LastID		uid	UID for non-byte tables, this is the last uid assigned for that table. (Read-only)
MinSize		uinteger_4	Number of rows initially requested for this table. The table can have the CreateRow method invoked on it this many times. This column is user-settable (access control permitting).
MaxSize		uint_4_def_0	Host-defined maximum number of rows for this table. The table will never have more than this many rows (though there are cases in which the created table will not be permitted by the system to have MaxSize rows). This column is user-settable (access control permitting), but the TPer may prevent the value in this column from being changed. A value of 0 indicates no host-defined limit of rows that may be created in this table.

The Table table contains one row for each table descriptor object, which store metadata about each of the tables in the SP.

In the Table table of every SP, there shall be a row for each table that is issued into that SP. Each of these rows shall have a CommonName column value. Each table at issuance shall have a CommonName column value of the Template from which it was issued – this is the name of the Template from the Admin SP's SPTemplates table.

In issued SPs (SPs other than the Admin SP), the TemplateID column value shall always be zeroes (a Null UID reference). In the Admin SP, the value of the TemplateID column may be zeroes. A value of zeroes in the TemplateID column of the Admin SP's Table table indicates that that row is active in the SP. Otherwise, the value of the TemplateID column in a row of the Admin SPs Table table shall be the uid of the row of the SPTemplates table to which that table belongs.

The Name-CommonName-TemplateID column value combination shall be unique for each row in the table.

5.3.2.4 Table and Method Metadata Group - Column (Array Table)

The table in this section describes the Metadata that the SP keeps about all of its tables.

Table 36 Column Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	UID of this row in the Column table. (Read-only)
Name	name	Name of the column. (Read-only)
CommonName	name	A name that may be shared among multiple columns. (Read-only)
Type	Type_ref	Type of the column's data. (Read-only)
IsIndex	boolean_def_false	The value of this column is True if this column is, or is part of, the unique index for the table. If the value of this column is False, this column is not a part of the table's index. (Read-only)
Byte	uinteger_4	Offset of column from start of row. (Read-only)
Transactional	boolean_def_true	Identifies if the column is subject to transactional rollback. (Read-only)
Next	Column_ref	Reference to the row of the Column table that represents the next column in this column's table. If this is the last column in the containing table, then the value of this column is 0. (Read-only)

The Column table has one row for every column of every table except byte tables.

If the value of the Transactional column is False, then modifications to this column take effect immediately, even if the method invocation that modifies the column is included in a transaction that has not yet resolved. Changes to the column are not rolled back if the transaction containing the modification is aborted. The value of this column for user-created table columns is True.

The value of the CommonName column for rows that exist upon issuance is the name of the Template (from the SPTemplates table) to which that column belongs.

The SP implementation is free to have hidden system columns in any table, as long as those columns do not interfere with host operations, including the operation of any methods invoked on that table. These columns shall not be recorded in the Column table.

5.3.2.5 Table and Method Metadata Group - Type (Object Table)

The Type table stores the information for all of the types used in the SP. All of the types predefined in the Core Spec shall be included by default in the table.

Table 37 Type Table Description

Column	IsIndex	Type	Description
UID		uid	The UID of the type. (Read-only)
Name	Yes	name	The name of the type. (Read-only)
CommonName	Yes	name	This is a name that may be shared by multiple types. (Read-only)
Format		type_def	This value will be 0 for a predefined type (integer, uinteger, bytes, max bytes). Otherwise this specifies the format of the type. For details, see the format specification, section 5.1.1. (Read-only)
Size		uinteger_2	Size (in bytes) needed to store a value of this type. (Read-only)
Default		ref_def_00	This column defines the default value for the type. (Read-only for pre-personalization types)

The `Type` table contains one row for each type in use in the SP. The host may add host-defined types by invoking the `CreateRow` method on the `Type` table.

No user-defined types shall be removed by the `Delete` or `DeleteRow` methods unless the TPer is able to verify that no column of that type is currently in use.

Types are often constructed of other types. The TPer shall prevent modification or removal of a type object upon which another type is dependent.

The TPer shall also prevent type recursion.

The size of the `Format` column is SSC-dependent. The value of the `Size` column includes any necessary overhead (such as for bytes{max=<n>} or for tagging a value of an `Alternative_Type`. The TPer calculates the value of this column. It is an error for the host to specify a value for this column in the `CreateRow` method invocation.

The `Default` column is used to identify the default value for that type. This default value is used when the `CreateRow` method is invoked and a column that uses that type does not have a value specified in the `CreateRow` invocation. The default value of the `Default` column is zeroes (a Null UID reference).

If the value of the `Default` column is zeroes then there shall be no default value for that type. A `CreateRow` invocation on a table with a column of that type shall have a value for that column specified in the method invocation, or the method invocation shall fail. Otherwise, the value of the `Default` column shall be a uidref to a byte table that contains the default value for the type. The value in the byte table shall be the same as required for messaging tokenization (see section 3.2.2.3), and shall be type checked by the TPer whenever a `CreateRow` is invoked that uses that value (i.e., that does not specify a value for a column of that type).

The format specification for specifying the value of the `Format` column is in section 5.1.1.

The `Type` table values that represent the built-in types, as well as all those types pre-defined in this specification, are be found in Table 30.

5.3.2.6 Table and Method Metadata Group - MethodID (Array Table)

This table associates method names and uids. Each value in the `Name` column must be unique. Life cycle permits this table to be read with the use of the Anybody authority, and prevents this table from being written by any authority.

Table 38 MethodID Table Description

Column	IsIndex	Type	Description

Column	IsIndex	Type	Description
RowNumber		uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID		uid	UID identifier of the method. (Read-only)
Name	Yes	name	Name of this method. (Read-only for pre-personalization methods)
CommonName	Yes	name	A name that may be shared among multiple methods. (Read-only for pre-personalization tables)
TemplateID	Yes	SPTemplates_ref	This is that Template's UID in the SPTemplates table. This may be zeroes in the Admin SP. (Read-only)

In the `MethodID` table of every SP, there shall be a row for each method that may be invoked on that SP. Each of these rows shall have a `CommonName` column value. Each row in the `MethodID` table shall have a `CommonName` column value of the Template from which it was issued. This is the name of the Template from the Admin SP's `SPTemplates` table.

In issued SPs (SPs other than the Admin SP), the `TemplateID` column value shall always be zeroes (a Null UID reference). In the Admin SP, the value of the `TemplateID` column may be zeroes. A value of zeroes in the `TemplateID` column of the Admin SP's `MethodID` table indicates that that row is active in the SP. Otherwise, the value of the `TemplateID` column in a row of the Admin SPs `MethodID` table shall be the uid of the row of the `SPTemplates` table to which that method belongs.

The `Name-CommonName-TemplateID` column value combination shall be unique for each row in the table.

5.3.2.7 Table and Method Metadata Group - Method (Array Table)

The table in this section describes the Metadata that the SP keeps about its SP/method, table/method, and object/method access control associations.

Table 39 Method Table Description

Column	IsIndex	Type	Description
RowNumber		uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID		uid	Unique identifier of this row in the Method table (Read-only)
InvokingID	Yes	table_object_ref	This is the uidref to the SP/Table/Object portion of this access control association. (Read-only)
MethodID	Yes	MethodID_ref	UID identifier for the method part of this access control association. (Read-only)
CommonName		name	A name that may be shared among multiple access control associations (Read-only)

Column	IsIndex	Type	Description
ACL		ACL	The ACL for this SP/method, table/method, or object/method combination. This column is modified/accessed via the methods GetACL, RemoveACE, and AddACE. This column shall not be modifiable directly via the Set method.
Log		log_select	Log whether this method succeeds, fails, or both (or neither). This column shall be disregarded if the Log Template has not been issued into the SP.
AddACEACL		ACL	This column holds the access control list that permits controls invocation of the AddACE method on the access control association represented by this row in the Method table.
RemoveACEACL		ACL	This column holds the access control list that permits controls invocation of the RemoveACE method on the access control association represented by this row in the Method table.
GetACLACL		ACL	This column holds the access control list that permits controls invocation of the GetACL method on the access control association represented by this row in the Method table.
DeleteMethodACL		ACL	This column holds the access control list that permits controls invocation of the DeleteMethod method on the access control association represented by this row in the Method table.
AddACELog		log_select	This column identifies the conditions under which logging of the AddACE method invocation on this access control association occurs. This column shall be disregarded if the Log Template has not been issued into the SP.
RemoveACELog		log_select	This column identifies the conditions under which logging of the RemoveACE method invocation on this access control association occurs. This column shall be disregarded if the Log Template has not been issued into the SP.
GetACLLog		log_select	This column identifies the conditions under which logging of the GetACL method invocation on this access control association occurs. This column shall be disregarded if the Log Template has not been issued into the SP.

Column	IsIndex	Type	Description
DeleteMethodLog		log_select	This column identifies the conditions under which logging of the DeleteMethod method invocation on this access control association occurs. This column shall be disregarded if the Log Template has not been issued into the SP.
LogTo		ref_def_00 {LogTableUID}	This identifies the log table to which log entries for this access control association are added. The default value of this column is 00s, which indicates that this access control association logs to the default log table. This column shall be disregarded if the Log Template has not been issued into the SP.

The `Method` table contains SP/method, table/method, and object/method access control associations and logging settings, and each access control association's related meta-ACL access requirements and meta-ACL logging settings.

New rows shall not be created in or deleted from the `Method` table directly. New rows are created in the `Method` table as a side effect whenever a table is created or when a row in an object table is created. `Method` table rows associated with a particular object or table are removed whenever that table or object is deleted.

5.3.2.8 Access Control Metadata Group - ACE (Object Table)

Table 40 ACE Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier of this ACE object (Read-only)
Name	Yes	name	Name of this ACE object (Read-only for pre-personalization ACEs)
CommonName	Yes	name	Name that may be shared among multiple ACE objects (Read-only for pre-personalization ACEs)
BooleanExpr		AC_element	A boolean expression of Authorities and/or Authority Classes that authorizes the ACE if true. If the conditions described in this access control element are true, then the ACE is considered authenticated.
RowStart		row_selection	The value of this column identifies the first row of the restriction that this ACE identifies. If the value of this column is 0, then this indicates the first row of the table.
RowEnd		row_selection	The value of this column identifies the last row of the restriction that this ACE identifies. This value shall be a higher value than RowStart. If the value of this column is 0, then this indicates the last row of the table.
ColStart		name	The value of this column identifies the first column of the restriction that this ACE identifies. Columns are ordered left to right in the order in which they appear in this specification. If the value of this column is a zero length bytes value, then this indicates the first column of the table.

Column	IsIndex	Type	Description
ColEnd		name	The value of this column identifies the last column of the restriction that this ACE identifies. Columns are ordered left to right in the order in which they appear in this specification. If the value of this column is a zero length bytes value, then this indicates the last column of the table.

The ACE table has one row for each access control element that may be authenticated by the host.

In the case of invoked SP methods, values of the RowStart, RowEnd, ColStart, and ColEnd columns of a referenced ACE object are ignored. In the case of table methods on object tables, or in the case of object methods, RowStart and RowEnd reference an object UID and should be equivalent to each other.

The values for RowStart and RowEnd must be applicable to the table upon which a method requiring authentication of this ACE is being invoked. If either the RowStart or RowEnd values are out of bounds for the table, then the invoked method shall fail and return an error. This same restriction applies to the ColStart and ColEnd column values as well.

5.3.2.9 Access Control Metadata Group - Authority (Object Table)

A Row of the Authority Table is called an Authority. An Authority is a specific use of a Credential and, possibly, other Authorities. A Class Authority is an authority object referenced by multiple Individual Authorities and does not use a Credential.

Table 41 Authority Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier of this authority. (Read-only)
Name	Yes	name	Name of this authority. (Read-only for pre-personalization authorities)
CommonName	Yes	name	Name common to several authorities. (Read-only for pre-personalization authorities)
IsClass		boolean	If True, this row is a class authority. If False, this row is an individual authority.
Class		Authority_ref	The value of this column designates the class to which this authority belongs.
Enabled		boolean	When this value is True, this Authority or Authority Class is Enabled. If the value of this column is False, then this authority is disabled.
Secure		messaging_type	This column identifies the type of secure messaging to be used
HashAndSign		hash_protocol	Identifies if hash/sign of session startup method parameters is required.
PresentCertificate		boolean	Determines if a certificate needs to be supplied with an authority at session startup
Operation		auth_method	The operation to perform with the Credential (e.g., Exchange, Signing, SymK, HMAC, PIN, None).
Credential		cred_object_uidref	This is the specific credential object to be used with this authority.

Column	IsIndex	Type	Description
ResponseSign		Authority_ref	This column identifies the signing authority with which the SP shall respond during session startup. This may be self-referential.
ResponseExch		Authority_ref	This column identifies the exchange authority with which the SP shall respond during session startup. This may be self-referential.
ClockStart		date	This value identifies the date on which this authority becomes valid.
ClockEnd		date	This value identifies the date on which this authority expires/becomes invalid.
Limit		uint_4_def_0	Sets a limit on the number of authentications for this authority.
Uses		uint_4_def_0	Total number of successful authentications with this authority, including both successful Session start-up invocations and Authenticate method invocations.
Log		log_select	These flags enable logging of different events that occur when attempting to authenticate this authority.
LogTo		ref_def_00	This identifies the log table to which log entries for operations with this authority are added.

The `Class` column identifies the authority class of which an authority object is a member. Class authorities may be members of another class authority. However, this shall only be valid if it extends to one level. Class authorities are not permitted to be members of a class authority that is already a member of another class authority. The TPer shall enforce this requirement. The value of this column is only valid if the value of the IsClass column is True. The value of this column shall be a Null UID reference if the authority is not a member of a class.

The `Enabled` column identifies if the authority object is active. All attempts to authenticate this authority either directly, through the use of the `Authenticate` method, or indirectly, as in during session startup, return an error if the value of this column is False. The default value of this column is True.

The `Secure` column identifies the type of secure messaging (if any) that is required by this authority, and identifies the size of the key(s) that shall be generated during secure session startup if confidential messaging is required. A value of “None” indicates secure messaging is not required and not permitted. The value of this column shall be enforced when any attempt is made to authenticate this authority, including the use of the `Authenticate` method. The options for this column, which are the options defined for the messaging_type type, are identified in Table 42.

Note that the IV size for both the CCM and GCM modes is 12-bytes. The lower 8-bytes are directly provided within the secure message. The upper 4-bytes of the IV are taken from the last 4-bytes of the EncryptSessionKey parameters of the `StartTrustedSession/SyncTrustedSession` method pair. See RFC 4106 (GCM) and RFC 4309 (CCM) for details. The EncryptSessionKey parameters of the `StartTrustedSession/SyncTrustedSession` method pair need to be 4 bytes longer for the CCM and GCM modes to accommodate the extra 4 bytes that are used as 'salt' within the IV.

Table 42 Secure Column Values

Column value	Algorithm	Secure Messaging Type
0	None	None
1	HMAC_SHA_256	Integrity only
2	HMAC_SHA_384	Integrity only

Column value	Algorithm	Secure Messaging Type
3	HMAC_SHA_512	Integrity only
4	RSASSA-PSS_1024 (PKCS #1 v1.5)	Integrity only
5	RSASSA-PSS_2048 (PKCS #1 v1.5)	Integrity only
6	RSASSA-PSS_3072 (PKCS #1 v1.5)	Integrity only
7	RSASSA-PSS_1024 (PKCS #1 v2.1)	Integrity only
8	RSASSA-PSS_2048 (PKCS #1 v2.1)	Integrity only
9	RSASSA-PSS_3072 (PKCS #1 v2.1)	Integrity only
10	ECDSA_256_SHA_256	Integrity only
11	ECDSA_384_SHA_384	Integrity only
12	ECDSA_512_SHA_512	Integrity only
13	CMAC_128 with 128-bit MAC	Integrity only
14	CMAC_256 with 128-bit MAC	Integrity only
15	GMAC_128 with 128-bit MAC and 96-bit IV	Integrity only
16	GMAC_256 with 128-bit MAC and 96-bit IV	Integrity only
17	AES_CBC_128	Confidentiality only
18	AES_CBC_256	Confidentiality only
19	AES_CBC_128 with HMAC_SHA_256	Integrity and Confidentiality
20	AES_CBC_256 with HMAC_SHA_256	Integrity and Confidentiality
21	AES_CBC_256 with HMAC_SHA_384	Integrity and Confidentiality
22	AES_CBC_256 with HMAC_SHA_512	Integrity and Confidentiality
23	AES_CCM_128 with 128-bit MAC	Integrity and Confidentiality
24	AES_CCM_256 with 128-bit MAC	Integrity and Confidentiality
25	AES_GCM_128 with 128-bit MAC	Integrity and Confidentiality
26	AES_GCM_256 with 128 bit MAC	Integrity and Confidentiality

The value of the `HashAndSign` column determines if hashing and signing of session startup method parameters is required. If the value of this column is other than “None”, a signed hash is to be used during session startup. The value of the `Operation` column and the type of the credential referenced in the `Credential` column (and the hash protocol identified in that credential) determine the type of the hashing and signing. Note that `HashAndSign` is only enforced for a particular authority during session startup. Otherwise, this attribute is ignored (for instance, during an `Authenticate` method invocation). For additional information see section 3.4.4.7 and section 5.3.4.1.4.

If the value of the `PresentCertificate` column is True, the authority is a public key authority, and the credential contains a certificate chain, then it shall be required that a certificate chain associated with this authority is sent as a parameter of the session startup protocol. If any of those conditions is False, no certificate is required to be sent. See the TCG Certificates Specification for more information on certificates.

The value of the `Credential` column identifies the specific credential object to be used with this authority. For a class authority, the value of this column shall be zeroes (a Null UID reference).

The value of the `ResponseSign` column identifies the authority with which the TPer shall respond in the `SyncSession` method of the session startup method exchange. The authority referenced in this column

identifies the authority to be used by the TPer as the SP Signing Authority. If the value of this column is 00s, then no SP Signing Authority shall be used for initiating that session.

The value of the ResponseExch column identifies the authority with which the TPer shall respond in the SyncSession method of the session startup method exchange. The authority referenced in this column identifies the authority to be used by the TPer as the SP Exchange Authority. If the value of this column is 00s, then no SP Exchange Authority shall be used for initiating that session.

An authority is automatically enabled starting on the date defined in the ClockStart column if the TPer has a trusted date. A value of all 0's indicates no start date, and the authority shall be authenticatable until the date in the ClockEnd column is reached. If the Clock Template has not been issued with this SP, then the value of this column shall be disregarded, and should be set to all zeroes. Any authority with a non-zero ClockStart date shall not be authenticatable if the ClockTime table's TrustMode column is "Timer".

An authority is automatically disabled starting on the date defined in the ClockEnd column if the TPer has a trusted date. A value of all zero indicates no end date, and the authority's ability to be authenticated shall not expire. If the Clock Template has not been issued with this SP, then the value of this column shall be disregarded, and should be set to all zeroes. Any authority with a non-zero ClockEnd date shall not be authenticatable if the ClockTime table's TrustMode column is "Timer"

The Limit column defines a limit on the number of times that an authority may be authenticated, either explicitly or implicitly. This value represents the maximum number of total successful authentications with this authority, including session start-up invocations and Authenticate method invocations. A value of 0 shall mean no limit. The default value of the Limit column is 0.

The value of the Uses column identifies the number of times an authority has been authenticated. If the value of Uses is equal to the value of Limit for this authority and the value of the Limit column is not 0, then this authority shall not be authenticatable, and attempts to authenticate shall result in an error response. This value is not subject to transactional rollbacks. The default value of the Uses column is 0.

The value of the Log column identifies when uses of this authority (i.e., authentications and authentication attempts) are logged. This logging is only applicable when authentications are done in establishing a session or in augmenting the authorities on it (via the Authenticate method), not when authentication is tested on a method. If the Log Template has not been issued into the SP, then this column is disregarded and should be set to zero.

The LogTo column identifies the Log table to which events related to this authority (session startups and authentications) are logged. The default value of this column is a Null UID reference, which indicates that this authority's operations log to the default log table (see section 5.7). This column shall be disregarded if the Log Template has not been issued into the SP.

5.3.2.10 Access Control Metadata Group - Certificates (Object Table)

Table 43 Certificates Table Description

Column	IsIndex	Type	Description
UID		uid	UID of this row of the Certificates table (Read-only)
Name	Yes	name	Name of this certificate
CommonName	Yes	name	Name that may be shared among multiple Certificates.
CertData		byte_table_ref	This is the uidref to the byte table that holds the certificate data for this Certificates object.
CertSize		uinteger_4	Number of bytes actually used in the certificate.

For composition and formatting of a certificate chain, see the TCG Certificates Specification.

5.3.2.11 Credential Table Group - C_PIN (Object Table)

Table 44 C_PIN Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier of this C_PIN object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization C_PIN objects)
CommonName	Yes	name	A name that may be shared among multiple C_PIN objects (Read-only for pre-personalization C_PIN objects)
PIN		password	Password string.
CharSet		ref_def_00	uidref to the byte table that holds the char set for the PIN. If the value of this column is zeroes, then the default character set is used with the GenKey method. (Read-only)
TryLimit		uinteger_4	Maximum number of failed tries before always failing.
Tries		uinteger_4	Current number of failed tries.
Persistence		boolean	Identifies if value of Tries column is persistent through power cycles

The `C_PIN` table contains one row for each password credential.

If the value of the `CharSet` column is zeroes (a Null UID reference), then the default character set used when creating a new `PIN` column value with the `GenKey` method shall be made up of the set of valid ASCII printable characters as defined in RFC 1345. The default value of the `CharSet` column is zeroes.

If not 00s, then the value of the `CharSet` column is a uidref to a byte table that contains the character set to be used when the `GenKey` method is invoked on this `C_PIN` object to generate a new password.

If the `CharSet` column value is not zeroes, it shall be a uid to a byte table in which shall be defined a character set to be used when creating a new `PIN` column value with the `GenKey` method. The character set defined in the byte table shall be made up of a subset of the set of valid ASCII printable characters as defined in RFC 1345.

The default value of the `TryLimit` column when a new `C_PIN` object is created is 0. The value 0 in this column indicates that there is no limit on the number of tries for that object.

The default value of the `Tries` column when a new `C_PIN` object is created is 0. If the value of the `TryLimit` column is not 0, then the value of the `Tries` column is incremented by the TPer on every failed `Authenticate`, including the implicit `Authenticate` if the authority is a Signing Authority invoked during session startup.

When the value of the `Tries` column is equal to the value of the `TryLimit` column, and the `TryLimit` column is not equal to 0, further attempts to authenticate using this credential will always fail (until the value of the `Tries` column is reset), but Tries will not increment beyond `TryLimit`.

The value of the `Tries` column is set to 0 by the TPer upon successful invocation of the `Authenticate` method or implicit session startup authentication of the authority referencing this `C_PIN` object.

The value of the `Tries` column may be reset from the host by successful invocation of the `Set` method on that cell to set the value to 0 (access control must be properly fulfilled).

Additionally, the value of the `Tries` column will be reset to 0 after a power cycle if the value of the `Persistence` column is False. Otherwise, the value of the `Tries` column will persist across power cycles.

If `TryLimit` is 0, there is no limit to the number of `Tries`, and `Tries` shall remain 0.

Note: The value of the `Tries` column is not subject to transactional rollback when changed by the TPer. The TPer shall be able to set the `Tries` column value during a Read-Only session, but the host shall only be able to set this column during a Read-Write session.

The C_PIN object with `UID=0x00 0x00 0x00 0x0B 0x00 0x00 0x00 0x01` and `Name="SID"` is the default SID object.

5.3.2.12 Credential Table Group - C_RSA_1024 (Object Table)

Table 45 C_RSA_1024 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among C_RSA_1024 objects (Read-only for pre-personalization objects)
Format		padding_type	This column defines the type of padding used with RSA encryption.
Pu_Exp		uinteger_128	RSA Public Exponent
Mod		uinteger_128	RSA Public Modulus
Pr_Exp		uinteger_128	RSA Private Exponent
P		uinteger_64	p and q, the primes from the key generation,
Q		uinteger_64	
Dmp1		uinteger_64	d mod (p-1) and d mod (q-1) (often known as dmp1 and dmq1)
Dmq1		uinteger_64	
Iqmp		uinteger_64	(1/q) mod p (often known as iqmp)
Hash		hash_protocol	If a referencing authority has a HashAndSign column value of True, this column identifies the hash algorithm to create the session startup method parameter MAC to be signed by this credential.
ChainLimit		int_1_def_0	The chaining limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.
Certificate		Certificates_ref	Certificate(s) – provides a chained set of unencoded X.509 certificates if needed to prove an ancestor authority

5.3.2.13 Credential Table Group - C_RSA_2048 (Object Table)

Table 46 C_RSA_2048 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only for pre-personalization objects)

Column	IsIndex	Type	Description
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_RSA_2048 objects (Read-only for pre-personalization objects)
Format		padding_type	This column defines the type of padding used with RSA encryption.
Pu_Exp		uinteger_256	RSA Public Exponent
Mod		uinteger_256	RSA Public Modulus
Pr_Exp		uinteger_256	RSA Private Exponent
p		uinteger_128	p and q, the primes from the key generation
q		uinteger_128	
Dmp1		uinteger_128	d mod (p-1) and d mod (q-1) (often known as dmp1 and dmq1)
Dmq1		uinteger_128	
Iqmp		uinteger_128	(1/q) mod p (often known as iqmp)
Hash		hash_protocol	If a referencing authority has a HashAndSign column value of True, this column identifies the hash algorithm to create the session startup method parameter MAC to be signed by this credential.
ChainLimit		int_1_def_0	The chaining limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

5.3.2.14 Credential Table Group - C_AES_128 (Object Table)

Table 47 C_AES_128 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_AES_128 objects (Read-only for pre-personalization objects)
Key		bytes_16	Key
Mode		symmetric_mode	Defines the mode with which this credential shall be used.
FeedbackSize		feedback_size	Feedback size for CFB mode
ResidualData		bytes_16	The value in this column provides the IV for the Encrypt/Decrypt method (unless the IV parameter in the EncryptInit/DecryptInit method is invoked).
Hash		hash_protocol	Defines the hash protocol to be used with this credential

The Mode column defines the encryption mode with which this credential shall be used. Valid values are ECB, CBC, CFB, OFB, GCM, CCM, CTR and MediaEncryption. MediaEncryption mode permits a vendor-specific encryption mode. Having a mode other than MediaEncryption does not prevent this credential from being used as a media encryption key. For additional information on media encryption, see 5.8.

The value in the ResidualData column provides the IV for the Encrypt/Decrypt method (unless the IV parameter in the EncryptInit/DecryptInit method is invoked). The TPer then sets this value as the last block encrypted by the Encrypt method or last block decrypted by the Decrypt method. Subsequent method invocations use this column value as its IV. The value set to this column during Encrypt/Decrypt operations is dependent on this object's mode, as defined in Table 48.

The Hash column defines the hash protocol to be used with this credential.

Table 48 C_AES_128 ResidualData Column Values

Mode	Column Value
ECB	All 00's
CBC	The ciphertext of the last block encrypted/decrypted
CFB	The (128 – FeedbackSize) LSBs of the last input to the Encrypt/Decrypt method, concatenated with the ciphertext of the last block encrypted/decrypted
OFB	The last output block of the Encrypt/Decrypt method
CTR	The last input block to the Encrypt/Decrypt method + 1
GCM	The last input block to the Encrypt/Decrypt method + 1
CCM	The last input block to the Encrypt/Decrypt method + 1
MediaEncryption	Vendor specific

5.3.2.15 Credential Table Group - C_AES_256 (Object Table)

Table 49 C_AES_256 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_AESS_256 objects (Read-only for pre-personalization objects)
Key		bytes_32	Key
Mode		symmetric_mode	Defines the mode with which this credential shall be used.
FeedbackSize		feedback_size	Feedback size for CFB mode
ResidualData		bytes_32	The value in this column provides the IV for the Encrypt/Decrypt method (unless the IV parameter in the EncryptInit/DecryptInit method is invoked).
Hash		hash_protocol	Defines the hash protocol to be used with this credential

The Mode column defines the encryption mode with which this credential shall be used. Valid values are ECB, CBC, CFB, OFB, GCM, CCM, CTR and MediaEncryption. MediaEncryption mode permits a vendor-specific encryption mode. Having a mode other than MediaEncryption does not prevent this credential from being used as a media encryption key. For additional information on media encryption, see 5.8.

The value in the `ResidualData` column provides the IV for the `Encrypt/Decrypt` method (unless the IV parameter in the `EncryptInit/DecryptInit` method is invoked). The TPer then sets this value as the last block encrypted by the `Encrypt` method or last block decrypted by the `Decrypt` method. Subsequent method invocations use this column value as its IV. The value set to this column during `Encrypt/Decrypt` operations is dependent on this object's mode, as defined in Table 50.

The `Hash` column defines the hash protocol to be used with this credential.

Table 50 C_AES_256 ResidualData Column Values

Mode	Column Value
ECB	All 00's
CBC	The ciphertext of the last block encrypted/decrypted
CFB	The (256 – FeedbackSize) LSBs of the last input to the Encrypt/Decrypt method, concatenated with the ciphertext of the last block encrypted/decrypted
OFB	The last output block of the Encrypt/Decrypt method
CTR	The last input block to the Encrypt/Decrypt method + 1
GCM	The last input block to the Encrypt/Decrypt method + 1
CCM	The last input block to the Encrypt/Decrypt method + 1
MediaEncryption	Vendor specific

5.3.2.16 Credential Table Group - C_EC_160 (Object Table)

Table 51 C_EC_160 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_EC_160 objects (Read-only for pre-personalization objects)
p		uinteger_20	Modulus
r		uinteger_20	Order of the curve
b		uinteger_20	Curve coefficient ($y^2=x^3-3x+b \bmod p$)
x		uinteger_20	Base point x-coordinate
y		uinteger_20	Base point y-coordinate
alpha		uinteger_20	Private key
u		uinteger_20	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_20	Public key y-coordinate: (u, v) = α (x,y)
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.

Column	IsIndex	Type	Description
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 52 represents the set of elliptic curve domain parameters as specified in AACS "Introduction and Common Cryptographic Elements". The entries p, r, b, x and y are represented in decimal format. These are example values for a curve that may be used with the `C_EC_160` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_160` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 52 AACS Values for C_EC_160

Column	Value
p	9008128236375876465141064625884558904987290070 71
r	9008128236375876465141065555665735887797707530 47
b	3663940346472317503243704002220025668443547038 32
x	2648656139597296470181136708546051628959770088 38
y	5184107595488316251041339274516893629618780869 7

5.3.2.17 Credential Table Group - C_EC_192 (Object Table)

Table 53 C_EC_192 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_EC_192 objects (Read-only for pre-personalization objects)
p		uinteger_24	Modulus
r		uinteger_24	Order of the curve
b		uinteger_24	Curve coefficient ($y^2=x^3-3x+b \bmod p$)
x		uinteger_24	Base point x-coordinate
y		uinteger_24	Base point y-coordinate
alpha		uinteger_24	Private key
u		uinteger_24	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_24	Public key y-coordinate: (u, v) = α (x,y)
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.

Column	IsIndex	Type	Description
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 54 represents the set of elliptic curve domain parameters that is the fixed set known as P-192 in FIPS 186-3 and secp192r1 in SEC2. The entries p, r, b, x and y represented in that table are example values for a curve that may be used with the `C_EC_192` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_192` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 54 FIPS P-192 Values for C_EC_192

Column	Value
p	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFF = $2^{192} - 2^{64} - 1$
r	FFFFFFFF FFFFFFFF FFFFFFFF 99DEF836 146BC9B1 B4D22831
b	64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1
x	188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012
y	07192B95 FFC8DA78 631011ED 6B24CDD5 73F977A1 1E794811

5.3.2.18 Credential Table Group - C_EC_224 (Object Table)

Table 55 C_EC_224 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_EC_224 objects (Read-only for pre-personalization objects)
p		uinteger_28	Modulus
r		uinteger_28	Order of the curve
b		uinteger_28	Curve coefficient ($y^2=x^3-3x+b \bmod p$)
x		uinteger_28	Base point x-coordinate
y		uinteger_28	Base point y-coordinate
alpha		uinteger_28	Private key
u		uinteger_28	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_28	Public key y-coordinate: (u, v) = α (x,y)
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.

Column	IsIndex	Type	Description
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 56 represents the set of elliptic curve domain parameters that is the fixed set known as P-224 in FIPS 186-3 and secp224r1 in SEC2. The entries p, r, b, x and y represented in that table are example values for a curve that may be used with the C_EC_224 table. These values are set as the default values for the associated columns when a new row is created in the C_EC_224 table and when values for those columns are not specified at table creation. These default values are not represented by a Type table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 56 FIPS P-224 Values for C_EC_224

Column	Value
p	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000000 00000000 00000001 = $2^{224} - 2^{96} + 1$
r	FFFFFFFF FFFFFFFF FFFFFFFF FFFF16A2 E0B8F03E 13DD2945 5C5C2A3D
b	B4050A85 0C04B3AB F5413256 5044B0B7 D7BFD8BA 270B3943 2355FFB4
x	B70E0CBD 6BB4BF7F 321390B9 4A03C1D3 56C21122 343280D6 115C1D21
y	BD376388 B5F723FB 4C22DFE6 CD4375A0 5A074764 44D58199 85007E34

5.3.2.19 Credential Table Group - C_EC_256 (Object Table)

Table 57 C_EC_256 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_EC_256 objects (Read-only for pre-personalization objects)
p		uinteger_32	Modulus
r		uinteger_32	Order of the curve
b		uinteger_32	Curve coefficient ($y^2=x^3-3x+b \bmod p$)
x		uinteger_32	Base point x-coordinate
y		uinteger_32	Base point y-coordinate
alpha		uinteger_32	Private key
u		uinteger_32	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_32	Public key y-coordinate: (u, v) = α (x,y)
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.

Column	IsIndex	Type	Description
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 58 represents the set of elliptic curve domain parameters is the fixed set known as P-256 in FIPS 186-3 and secp256r1 in SEC2. The entries p, r, b, x and y represented in that table are example values for a curve that may be used with the C_EC_256 table. These values are set as the default values for the associated columns when a new row is created in the C_EC_256 table and when values for those columns are not specified at table creation. These default values are not represented by a Type table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 58 FIPS P-256 Values for C_EC_256

Column	Value
p	FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFF = $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
r	FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551
b	5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B
x	6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296
y	4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5

5.3.2.20 Credential Table Group - C_EC_384 (Object Table)

Table 59 C_EC_384 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_EC_384 objects (Read-only for pre-personalization objects)
p		uinteger_48	Modulus
r		uinteger_48	Order of the curve
b		uinteger_48	Curve coefficient ($y^2=x^3-3x+b \bmod p$)
x		uinteger_48	Base point x-coordinate
y		uinteger_48	Base point y-coordinate
alpha		uinteger_48	Private key
u		uinteger_48	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_48	Public key y-coordinate: (u, v) = α (x,y)

Column	IsIndex	Type	Description
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 60 represents the set of elliptic curve domain parameters is the fixed set known as P-384 in FIPS 186-3 and secp384r1 in SEC2. The entries p, r, b, x and y represented in that table are example values for a curve that may be used with the C_EC_384 table. These values are set as the default values for the associated columns when a new row is created in the C_EC_384 table and when values for those columns are not specified at table creation. These default values are not represented by a Type table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 60 FIPS P-384 Values for C_EC_384

Column	Value
p	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF 00000000 00000000 FFFFFFFF = $2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
r	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF C7634D81 F4372DDF 581A0DB2 48B0A77A ECEC196A CCC52973
b	B3312FA7 E23EE7E4 988E056B E3F82D19 181D9C6E FE814112 0314088F 5013875A C656398D 8A2ED19D 2A85C8ED D3EC2AEF
x	AA87CA22 BE8B0537 8EB1C71E F320AD74 6E1D3B62 8BA79B98 59F741E0 82542A38 5502F25D BF55296C 3A545E38 72760AB7
y	3617DE4A 96262C6F 5D9E98BF 9292DC29 F8F41DBD 289A147C E9DA3113 B5F0B8C0 0A60B1CE 1D7E819D 7A431D7C 90EA0E5F

5.3.2.21 Credential Table Group - C_EC_521 (Object Table)

Table 61 C_EC_521 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_EC_521 objects (Read-only for pre-personalization objects)
p		uinteger_66	Modulus
r		uinteger_66	Order of the curve
b		uinteger_66	Curve coefficient ($y^2=x^3-3x+b \bmod p$)
x		uinteger_66	Base point x-coordinate
y		uinteger_66	Base point y-coordinate

Column	IsIndex	Type	Description
alpha		uinteger_66	Private key
u		uinteger_66	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_66	Public key y-coordinate: (u, v) = α (x,y)
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 62 represents the set of elliptic curve domain parameters is the fixed set known as P-521 in FIPS 186-3 and secp521r1 in SEC2. The entries p, r, b, x and y represented in that table are example values for a curve that may be used with the C_EC_521 table. These values are set as the default values for the associated columns when a new row is created in the C_EC_521 table and when values for those columns are not specified at table creation. These default values are not represented by a Type table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 62 FIPS P-521 Values for C_EC_521

Column	Value
p	01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF = 2^{521} - 1
r	01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8 899C47AE BB6FB71E 91386409
b	0051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00
x	00C6 858E06B7 0404E9CD 9E3ECB66 2395B442 9C648139 053FB521 F828AF60 6B4D3DBA A14B5E77 EFE75928 FE1DC127 A2FFA8DE 3348B3C1 856A429B F97E7E31 C2E5BD66
y	0118 39296A78 9A3BC004 5C8A5FB4 2C7D1BD9 98F54449 579B4468 17AFBD17 273E662C 97EE7299 5EF42640 C550B901 3FAD0761 353C7086 A272C240 88BE9476 9FD16650

5.3.2.22 Credential Table Group - C_EC_163 (Object Table)

Table 63 C_EC_163 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)

Column	IsIndex	Type	Description
CommonName	Yes	name	A name that may be shared among multiple C_EC_163 objects (Read-only for pre-personalization objects)
k1		uinteger_1	High non-leading, non-constant term of irreducible pentanomial
k2		uinteger_1	Middle non-leading, non-constant term of irreducible pentanomial
k3		uinteger_1	Low non-leading, non-constant term of irreducible pentanomial
r		uinteger_21	Order of the curve
a		uinteger_1	Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$), must be zero or one
b		uinteger_21	Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$)
x		uinteger_21	Base point x-coordinate
y		uinteger_21	Base point y-coordinate
alpha		uinteger_21	Private key
u		uinteger_21	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_21	Public key y-coordinate: (u, v) = α (x,y)
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 64 represents the set of elliptic curve domain parameters that is the fixed set known as K-163 in FIPS 186-3 and sect163k1 in SEC2. The entries k1, k2, k3, r, a, b, x and y represented in that table are example values for a curve that may be used with the `C_EC_163` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_163` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 64 FIPS K-163 Values for C_EC_163

Column	Value
k1	07
k2	06
k3	03
r	04 00000000 00000000 00020108 A2E0CC0D 99F8A5EF
a	01
b	00 00000000 00000000 00000000 00000000 00000001
x	02 FE13C053 7BBC11AC AA07D793 DE4E6D5E 5C94EEE8
y	02 89070FB0 5D38FF58 321F2E80 0536D538 CCDAA3D9

5.3.2.23 Credential Table Group - C_EC_233 (Object Table)

Table 65 C_EC_233 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_EC_233 objects (Read-only for pre-personalization objects)
k		uinteger_2	Non-leading, non-constant term of irreducible trinomial
r		uinteger_30	Order of the curve
a		uinteger_1	Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$), must be zero or one
b		uinteger_30	Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$)
x		uinteger_30	Base point x-coordinate
y		uinteger_30	Base point y-coordinate
alpha		uinteger_30	Private key
u		uinteger_30	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_30	Public key y-coordinate: (u, v) = α (x,y)
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 66 represents the set of elliptic curve domain parameters that is the fixed set known as K-233 in FIPS 186-3 and sect233k1 in SEC2. The entries k, r, a, b, x and y represented in that table are example values for a curve that may be used with the `C_EC_233` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_233` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 66 FIPS K-233 Values for C_EC_233

Column	Value
k	4A (= 74 in decimal)
r	0080 00000000 00000000 00000000 00069D5B B915BCD4 6EFB1AD5 F173ABDF
a	00
b	0000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
x	0172 32BA853A 7E731AF1 29F22FF4 149563A4 19C26BF5 0A4C9D6E EFAD6126
y	01DB 537DECE8 19B7F70F 555A67C4 27A8CD9B F18AEB9B 56E0C110 56FAE6A3

5.3.2.24 Credential Table Group - C_EC_283 (Object Table)

Table 67 C_EC_283 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_EC_283 objects (Read-only for pre-personalization objects)
k1		uinteger_1	High non-leading, non-constant term of irreducible pentanomial
k2		uinteger_1	Middle non-leading, non-constant term of irreducible pentanomial
k3		uinteger_1	Low non-leading, non-constant term of irreducible pentanomial
r		uinteger_36	Order of the curve
a		uinteger_1	Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$), must be zero or one
b		uinteger_36	Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$)
x		uinteger_36	Base point x-coordinate
y		uinteger_36	Base point y-coordinate
alpha		uinteger_36	Private key
u		uinteger_36	Public key x-coordinate: (u, v) = α (x,y)
v		uinteger_36	Public key y-coordinate: (u, v) = α (x,y)
Hash		hash_protocol	The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority requires HashAndSign for session startup methods.
ChainLimit		integer_1	The chaining Limit for using a chained down key from this one. –1 indicates no limit. 0, no chain, is the default.
Certificate		Certificates_ref	Certificate(s) – provides a (possibly chained) set of unencoded X.509 certificates if needed to prove signing from an ancestor authority

Table 68 represents the set of elliptic curve domain parameters that is the fixed set known as K-283 in FIPS 186-3 and sect283k1 in SEC2. The entries k1, k2, k3, r, a, b, x and y represented in that table are example values for a curve that may be used with the `C_EC_283` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_283` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry – the TPer shall be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 68 FIPS K-283 Values for C_EC_283

Column	Value
k1	0C (= 12 in decimal)
k2	07
k3	05

Column	Value
r	01FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFE9AE 2ED07577 265DFF7F 94451E06 1E163C61
a	00
b	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
x	0503213F 78CA4488 3F1A3B81 62F188E5 53CD265F 23C1567A 16876913 B0C2AC24 58492836
y	01CCDA38 0F1C9E31 8D90F95D 07E5426F E87E45C0 E8184698 E4596236 4E341161 77DD2259

5.3.2.25 Credential Table Group – C_HMAC_160 (Object Table)

Table 69 C_HMAC_160 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_HMAC_160 objects (Read-only for pre-personalization objects)
Key		bytes_20	Key
Hash		hash_protocol	Defines the hash protocol to be used with this credential

The value of the `Key` column of this table holds key material to be used with an HMAC authentication operation or a host-invoked HMAC operation (as enabled by the Crypto Template).

The value of the `Hash` column identifies the hash protocol to be used with this HMAC credential when this credential is referenced by an authority and used for authentication. The value of this column is ignored for host-invoked HMAC operations (see the Crypto Template section for additional details).

See FIPS-198 for details on matching key size to hash protocol selection.

5.3.2.26 Credential Table Group – C_HMAC_256 (Object Table)

Table 70 C_HMAC_256 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_HMAC_256 objects (Read-only for pre-personalization objects)
Key		bytes_32	Key
Hash		hash_protocol	Defines the hash protocol to be used with this credential

The value of the `Key` column of this table holds key material to be used with an HMAC authentication operation or a host-invoked HMAC operation (as enabled by the Crypto Template).

The value of the Hash column identifies the hash protocol to be used with this HMAC credential when this credential is referenced by an authority and used for authentication. The value of this column is ignored for host-invoked HMAC operations (see the Crypto Template section for additional details).

See FIPS-198 for details on matching key size to hash protocol selection.

5.3.2.27 Credential Table Group – C_HMAC_384 (Object Table)

Table 71 C_HMAC_384 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_HMAC_384 objects (Read-only for pre-personalization objects)
Key		bytes_48	Key
Hash		hash_protocol	Defines the hash protocol to be used with this credential

The value of the Key column of this table holds key material to be used with an HMAC authentication operation or a host-invoked HMAC operation (as enabled by the Crypto Template).

The value of the Hash column identifies the hash protocol to be used with this HMAC credential when this credential is referenced by an authority and used for authentication. The value of this column is ignored for host-invoked HMAC operations (see the Crypto Template section for additional details).

See FIPS-198 for details on matching key size to hash protocol selection.

5.3.2.28 Credential Table Group – C_HMAC_512 (Object Table)

Table 72 C_HMAC_512 Table Description

Column	IsIndex	Type	Description
UID		uid	This is the unique identifier for this object. (Read-only)
Name	Yes	name	This is the name of this object. (Read-only for pre-personalization objects)
CommonName	Yes	name	A name that may be shared among multiple C_HMAC_512 objects (Read-only for pre-personalization objects)
Key		bytes_64	Key
Hash		hash_protocol	Defines the hash protocol to be used with this credential

The value of the Key column of this table holds key material to be used with an HMAC authentication operation or a host-invoked HMAC operation (as enabled by the Crypto Template).

The value of the Hash column identifies the hash protocol to be used with this HMAC credential when this credential is referenced by an authority and used for authentication. The value of this column is ignored for host-invoked HMAC operations (see the Crypto Template section for additional details).

See FIPS-198 for details on matching key size to hash protocol selection.

5.3.3 Methods

This section details the methods provided to an SP by the Base Template.

5.3.3.1 SP Method Group - DeleteSP (Method)

```
SPUID.DeleteSP[]
=>
[ Result : boolean ]
```

This method is used to delete the SP to which the DeleteSP method has been invoked.

The TPer owner is able to delete an SP by opening a session to the Admin SP and invoking the Delete method on the SP object in the Admin SP's SP table. However, the SP owner probably cannot delete the SP in this way, and instead uses this method.

This method operates within a Read-Write session to the SP that is being deleted. The SP will not be deleted until the session is successfully closed. Upon successful deletion of the SP, the following changes are made:

- The row in the Admin SP's SP table that represents this SP is deleted.
- The value of the Instances column of the Admin SP's Template table is reduced by 1 for each of the Templates that had been issued into the SP being deleted.
- The SP itself is deleted. The means of deletion is implementation-specific. Once the SP has been deleted, the Host shall no longer have the capability to open sessions to the SP.
- Any TPer functionality affected by the existence of the SP based on the templates incorporated into it is modified as defined in the appropriate Template reference section of the Core Spec.

Since an SP may be disabled by the SP owner or frozen by the TPer owner, this method shall be invokable on a disabled/frozen SP, as the entity that caused entry to the disabled or frozen state may not be available or have appropriate permission to reenable/unfreeze the SP prior to deletion (access control must still be fulfilled). See section 4

This method shall only be successfully invoked in a Read-Write session.

5.3.3.1.1 Fails

- If the method is invoked from within a Read-Only session.

5.3.3.2 Basic Table Method Group - CreateTable (SP Method)

```
SPUID.CreateTable[
       NewTableName : name,
       Kind : table_kind,
       GetSetACL : ACL,
       Columns : columns,
       MinSize : uinteger_4,
       MaxSize = uinteger_4,
       HintSize = uinteger_4,
       CommonName = name]
=>
[ UID : uid, Rows : uinteger_4 ]
```

This method is used to create a new table in an SP. For a byte table, all rows exist at table creation. For the other table types, no rows will exist, but are inserted using the CreateRow method.

The **NewTableName** parameter is the name for this table. The NewTableName-CommonName combination shall be unique within the Table table.

The **Kind** parameter identifies the table's type (object, array, or byte).

GetSetACL is the list of ACEs placed in the access control lists of the GetACL, AddACE, and RemoveACE methods for the methods available on the new table.

The **Columns** parameter defines the columns of the new table. For byte tables this parameter must be an empty list.

The **MinSize** parameter is used to define the initial number of rows allocated for the new table.

The optional **MaxSize** parameter defines the host-requested maximum number of rows that can be created for the table.

The optional **HintSize** parameter is used to suggest a number of rows to be created for the table.

The method returns the UID of the table, and the number of rows allocated.

5.3.3.2.1 Fails

- If a table with the specified name already exists.
- If there isn't space in the SP for the new table.
- If metadata/support tables (i.e. Table, Column, Method, and ACE) are not all able to create all required rows to support this table.
- If TPer determines MinSize is too large.

5.3.3.3 Basic Table Method Group - Delete (Object Method)

```
ObjectUID.Delete[ ]
=>
[ Result : boolean ]
```

Successful invocation of this method deletes the object upon which this method was invoked.

Upon successful deletion of an object, rows in the `Method` table where this object's UID appears in the `Type` column shall be deleted.

In the case of successful invocation of this method on a table descriptor object (a row in the `Table` table), the associated table is deleted. The rows in the `Method` table where the table's or table descriptor object's UID appear in the `InvokingID` column are deleted. The rows in the `Column` table that are associated with the table are also deleted.

If invoked on an SP object (a row in the Admin SP's `SP` table), the SP is deleted. Deleting an SP in this fashion has the same effects as detailed in 5.3.3.1.

5.3.3.3.1 Fails

- If the object does not exist.

5.3.3.4 Basic Table Method Group - CreateRow (Table Method)

```
TableUID.CreateRow[
       Data : row_data ]
=>
[ Result : createrow_result ]
```

This method inserts one or multiple rows into a table. This method is not available on byte tables. The list of refs and/or uidrefs returned is the list of all row references and/or UIDs of the rows created.

If a row with the specified indexed column values already exists in the table in which the new row is being created, it is deleted before the new row is inserted. Access control must be satisfied on the `Delete` or `DeleteRow` method for that row for the "overwrite" to occur. If ACLs do not permit the row to be deleted, an error is returned and the table is unchanged (no new row is created).

When a row in an object table is created, a number of `Method` table rows are created that correspond to the default methods permitted for the created object. ACLs are set on those methods, and in the meta-ACL methods associated with those methods, as follows:

- Using the HostSigningAuthority from the `StartSession` method, if provided.
- Otherwise, using the HostExchangeAuthority from the `StartSession` method, if provided.
- Otherwise, using the Anybody Authority.

A value for a row's `UID` column, or any other system column, cannot be specified.

Note that `CreateRow` may be limited in some instances based on required default values for some table columns, and may be required to make certain validity checks when creating rows for some tables. These instances will be called out in the pertinent Template Reference sections.

5.3.3.4.1 Fails

- When the table is full (i.e. MaxSize of the table was reached).
- If a row where the indexed column value combination already exists that is the same as that requested in the method cannot be over-written
- Columns specified are not part of table definition.
- Attempts to create more rows than may be allocated
- If all required associated rows are not able to be created in all related tables (i.e. the Table, Method, Column, and ACE tables)

5.3.3.5 Basic Table Method Group - DeleteRow (Table Method)

```
TableUID.DeleteRow[
       Where : row_selection,
       Count = uinteger_4 ]
=>
[ Result : boolean ]
```

This method is used to delete table rows. This method shall not be able to be successfully invoked on byte tables.

Invoking this method deletes **Count** rows, beginning with the row addressed by the value in the **Where** parameter.

If not provided, Count shall default to 1.

Count is not permitted for object tables (adjacency has no meaning), and shall be disregarded by the TPer. Only one row in an object table at a time may be deleted by invocation of this method – that row is identified in the row_selection parameter.

For side effects of deleting objects, see the description of the `Delete` method (section 5.3.3.3).

5.3.3.5.1 Fails

- If the addressed row does not exist.
- If Count is specified as 0.
- If Count is specified and there are not Count rows starting at Where.

5.3.3.6 Basic Table Method Group - Get (Table and Object Method)

```
TableUID.Get[
ObjectUID.Get[
       Cellblock : cell_block ]
=>
[ Result : get_result ]
```

This method is used to fetch the values of selected table cells.

The **Cellblock** parameter defines the scope of the data that the method is attempting to retrieve by identifying the rectangular range of cell values on which the method should operate.

Successful invocation of this method shall only return the values that are readable based on the currently authenticated authorities and ACE restrictions for this method. It is not an error to request columns that are restricted by an authenticated ACE.

If multiple row values are returned from an array table, rows shall be returned from the lowest numbered row to the highest numbered row. Column name-value pairs shall be returned in the order in

which they are listed in the Column table, with the first column linked from the Table's table descriptor object in the Table table. This first column links to the second column in its Next column, and so forth.

When an ACE with row and column restrictions is used on a table that puts the row or column restrictions out of bounds for that table, an authentication failure error is returned.

If the method is invoked on a byte table, the return type is ByteColumns. If the method is invoked on a non-byte table, the return type is Columns.

5.3.3.6.1 Fails

- If table/object doesn't exist.
- If the object method's Cellblock parameter contains row values.
- If the method is invoked on a Byte table and has column values in the Cellblock parameter.
- If the any of the Cellblock parameter values are out of bounds for the table upon which it was invoked.

5.3.3.7 Basic Table Method Group - Set (Table and Object Method)

```
TableUID.Set[
ObjectUID.Set[
       Where : cell_block,
       Values : set_values ]
=>
[ Result : boolean ]
```

This method is used to change the values of selected table cells. Unlike with the Get method, if any of the cells cannot be written, Set shall return an error. Either all the changes will be made, or none of them will be made.

The **Where** parameter defines the location of the cells whose values the method is attempting to change. It is an error for the object method's Where parameter to contain row information. Attempting to invoke the Set method on a range of objects (a range of rows in an object table) with the table method shall result in an error.

5.3.3.7.1 Fails

- If the table/object doesn't exist.
- If an attempt is made to change the value of an UID or other system cell.
- If an attempt is made to set a cell to a value larger than that cell's type allows.
- If the method is invoked on a byte table and the cell_block parameter contains Column values
- Set is restricted by an access control limitation on any of the rows and columns requested.

5.3.3.8 Basic Table Method Group - Next (Table Method)

```
TableUID.Next[
       Where = row_selection,
       Count = uinteger_4 ]
=>
[ Result : next_result ]
```

When successfully invoked on an array table, the Next method returns zero or more row number/uidref pairs currently in use in the table following the specified **Where** row, iterating sequentially (by RowNumber column value) through the table rows. If Where is not specified, the first row of the table is the first row number returned. If **Count** is not specified, it defaults to 1. If there are fewer than Count rows defined after the indicated starting row, only the defined row numbers are returned.

Since object tables are unordered, the iteration that results from successful invocation of this method on an object table will cause the method to "visit" the rows in the table in some undefined order. The method invocation returns zero or more uidrefs "following" the specified Where row, iterating through the table. If a value for the Where parameter is not specified in the method invocation, iteration starts at the "beginning" of the table.

The list of returned uidrefs that result from invocation of the `Next` method on an object table returns those uidrefs in an arbitrary order. Results are not guaranteed to be consistent if the object table is modified between calls to `Next`. The implementation is required to visit all rows of an object table only if the table is not changed during the iteration.

5.3.3.8.1 Fails

- If the table/object doesn't exist.

5.3.3.9 Basic Table Method Group - GetFreeSpace (SP Method)

```
SPUID.GetFreeSpace[]
=>
[uinteger_8, Table_ref_rows_list]
```

The `GetFreeSpace` method is an SP method that enables the host to retrieve the number of rows that can be additionally created in each table.

Invoking `GetFreeSpace` returns two values. The first return value is the approximate amount of free space (in bytes) available in the SP. The second is a list containing the UID of each table descriptor object and the number of rows that can be additionally created for each table (separately) under current conditions of the SP and the TPer. This number may change in subsequent invocations of this method, based on modifications subsequent to the method invocation.

The number of rows returned for a table(s) is not directly related to the free space remaining on the SP. The number of rows is only indicative of how many rows the system can generate per table.

5.3.3.10 Basic Table Method Group - GetFreeRows (Object Method)

```
TableObjectUID.GetFreeRows[]
=>
[ uinteger_4 ]
```

The `GetFreeRows` method is a table method that enables the host to retrieve the number of rows that may be additionally created in a table.

When `GetFreeRows` is invoked, the TPer returns only the number of rows that can be additionally created for that table.

The number of rows returned for a table(s) is not directly related to the free space remaining on the SP. The number of rows is only indicative of how many rows the system can generate per table.

5.3.3.10.1 Fails

- When the table TableObjectUID does not exist in the SP.

5.3.3.11 Method Manipulation Group - DeleteMethod (Meta-Method)

```
MethodTableUID.DeleteMethod[
      InvokingID : table_object_ref,
      MethodID : MethodID_ref ]
=>
[ Result : boolean ]
```

Successful invocation of the `DeleteMethod` method removes the indicated SP/method, table/method, or object/method access control association from the `Method` table. The association that is deleted from the `Method` table is the row where the `InvokingID` column value is the **InvokingID** parameter of the method, and the value of the `MethodID` column is the uid referenced in the **MethodID** parameter of the `DeleteMethod` invocation.

The `DeleteMethod` method is typically used during personalization, and allows the personalizing host to prevent the usage of certain methods on certain tables, objects, or the SP by removing the access control association that permits the method to be invoked.

This does not remove the capability of invoking the indicated method from the SP entirely. It only removes the indicated access control association that allows the method to be invoked in that particular fashion.

5.3.3.11.1 Fails

- If the Type/Method combination does not exist.

5.3.3.12 Access Control Method Group - Authenticate (SP Method)

```
SPUID.Authenticate[
       Authority : Authority_ref,
       Challenge = challenge ]
=>
[ typeOr{ Success : boolean,
Response : response } ]
```

Authorities invoked during session startup are implicitly authenticated. The `Authenticate` method is used to explicitly authenticate an authority within a session, i.e., after a session has already successfully begun.

The implementation may limit the number of authorities that may be authenticated at any one time (as recorded in the MaxAuthentications value of the `Properties` method). If the authentication attempt would cause the MaxAuthentications property value to be exceeded for the session, a properly invoked `Authenticate` method shall return a status of SUCCESS and a result of False.

5.3.3.12.1 Fails

- If the authority called out in the method invocation does not exist.
- If the secure messaging required by the authority is not in effect.
- If the Challenge in the first `Authenticate` method invocation does not match that expected by the TPer.
- If the Challenge returned in the second `Authenticate` invocation does not match that expected by the TPer.

5.3.3.13 Access Control Method Group - GetACL (Meta-Method)

```
MethodTableUID.GetACL[
       InvokingID : table_object_ref,
       MethodID : MethodID_ref ]
=>
[ ACL : ACL ]
```

This method is used to retrieve the contents of an access control association's ACL, which are stored in the `Method` table. This method returns a response of type "ACL", which is a list of uidrefs to ACE objects.

The **InvokingID** parameter is the uidref to this SP (always 00 00 00 00 00 00 00 01), the table, or the object of the access control association.

The **MethodID** parameter is the uidref to the method of the access control association. This is the uidref of the method object in the `MethodID` table.

5.3.3.13.1 Fails

- If the Type/Method combination does not exist.

5.3.3.14 Access Control Method Group - AddACE (Meta-Method)

```
MethodTableUID.AddACE[
       InvokignID : table_object_ref,
       MethodID : MethodID_ref,
       ACE : ACE_table_ref ]
=>
[ Result : boolean ]
```

This method is used to add an ACE to an existing SP/method, table/method, or object/method access control association, which is a row in the `Method` table.

The **InvokingID** parameter is the uidref to this SP (always 00 00 00 00 00 00 00 01), the table, or the object of the access control association.

The **MethodID** parameter is the uidref to the method of the access control association. This is the uidref of the method object in the `MethodID` table.

The **ACE** parameter is a uidref to the ACE to be added to the ACL column of the appropriate `Method` table row.

5.3.3.14.1 Fails

- If the Type/Method combination does not exist.
- If the ACE does not exist in the ACE table.
- If the ACE already exists in the ACL of the invoked access control association.
- If the ACL of the invoked access control association is full.

5.3.3.15 Access Control Method Group - RemoveACE (Meta-Method)

```
MethodTableUID.RemoveACE[
       InvokingID : table_object_ref,
       MethodID : MethodID_ref,
       ACE : ACE_table_ref]
=>
[ Result : boolean ]
```

This method is used to remove an ACE from an ACL in an existing SP/method, table/method, or object/method access control association, which are rows in the `Method` table.

The **InvokingID** parameter is the uidref to this SP (always 00 00 00 00 00 00 00 01), the table, or the object of the access control association.

The **MethodID** parameter is the uidref to the method of the access control association. This is the uidref of the method object in the `MethodID` table.

The **ACE** parameter is a uidref to the ACE to be removed from the ACL column of the appropriate `Method` table row.

5.3.3.15.1 Fails

- If the Type/Method combination does not exist.
- If the ACE does not exist in the ACE table.

5.3.3.16 Key Related Method Group - GenKey (Object Method)

This section describes the method used for key creation.

```
CredentialObjectUID.GenKey[
       PublicExponent = uinteger_4,
       PinLength = uinteger_1]
=>
[ Result : boolean ]
```

An existing `Credential` object is filled in with new key material. This method fills in the new key as appropriate for the type of the credential on which the method was invoked.

If this method is invoked on an RSA object (`C_RSA_1024` or `C_RSA_2048`) and the optional **PublicExponent** parameter is not specified, then the keys shall be calculated using the public exponent 2^16+1 (65537). The key randomly generated with this method for RSA keys is a 1 followed by n-1 random bytes, where n is the size of the key.

If this method is invoked on a `C_PIN` object, then a new value with **PinLength** characters is generated and stored in that `C_PIN` object's `Password` column. The character set used to generate the `C_PIN` value is referenced in the `C_PIN` table's `CharacterSet` column, or the default character set if the `C_PIN` table's `CharacterSet` column is 00s (see C_PIN table description in section 5.3.2.11).

If PinLength is not specified in the method invocation, the default value is 32. The maximum permitted value for the PinLength parameter is 32. Successful invocation of this method on a `C_PIN` object sets the value of that object's `Tries` column to 0.

5.3.3.16.1 Fails

- If the credential object does not exist.
- If a bad exponent is included.
- If PinLength is greater than the size of the `Password` column in the `C_PIN` table.

5.3.4 Description

5.3.4.1 Authentication

5.3.4.1.1 Credential Tables

Credential tables represent an extensible basis for providing the public and private parts of authentication mechanisms and key stores. Each credential table represents a different mechanism or key store type and each row a different authentication or key using the mechanism or key store represented by the table. The credential tables supported shall be reflected in the `CryptoSuite` table in the Admin SP.

Credential tables contain secrets that might need to never leave the TPer. Normal ACLs can prevent that in the case of an attack that comes in over the interface. To help protect against an attack in which the TPer electronics are changed, some column values may be hidden in storage by the TPer. In the credential table definitions, marked columns (those shaded gray) may be hidden. The hiding of those columns and the means by which they are hidden are implementation-specific.

5.3.4.1.2 Authorities

Authorities are objects in the `Authority` table. Each authority object is made up of columns that identify the authentication method for that authority, the necessary credentials for authenticating, and whether secure messaging is required during session startup for, or authentication of, that authority.

Authorities are made up of two types – Class and Individual.

Class authorities are a convenient grouping method provided to simplify simultaneous modification of multiple access control elements. Class authorities may be members of another class authority, but this shall not be permitted to expand beyond a single level. The TPer shall enforce that class authorities are not permitted to be members of a class authority that is already a member of another class authority. Class authorities shall not be permitted to reference credentials or secure messaging requirements. Class authorities cannot be directly authenticated – authentication attempts that reference class authorities shall always fail.

Individual authorities may be members of class authorities. Each individual authority shall only be a member of a single class. When an individual authority is authenticated, either from session startup or explicitly via the `Authenticate` method, the class authority that the individual authority references is considered to be authenticated also. If that class authority also references a class, then the class authority referenced by the initial class authority is also considered to be authenticated.

The authorities required by the Base Template are enumerated in Table 73.

Table 73 Default Base Template Authorities

Name	UID	Common Name	IsClass	Class	Operation
Anybody	00 00 00 09 00 00 00 01	Anybody	False		Sign
Admins	00 00 00 09 00 00 00 02	Admin	True		
Makers	00 00 00 09 00 00 00 03	Maker	True		
MakerSymK	00 00 00 09 00 00 00 04	Maker	False	Makers	SymK
MakerPuK	00 00 00 09 00 00 00 05	Maker	False	Makers	Sign
SID	00 00 00 09 00 00 00 06	TPerOwner	False		Password
TPerSign	00 00 00 09 00 00 00 07	TPerSign	False		TPerSign
TPerExch	00 00 00 09 00 00 00 08	TPerExch	False		TPerExchange
AdminExch	00 00 00 09 00 00 00 09	Admin	False	Admins	Exchange

The Anybody authority may used as the Host Signing Authority during session startup, and when used in this way allows session startup without providing a proof or secret. If a value is included in the `StartSession` method's HostChallenge parameter where the Anybody authority is called out as the HostSigningAuthority, the HostChallenge parameter shall be ignored by the TPer.

The Anybody authority is always considered "authenticated" within a session, even if the Anybody authority was not specifically called out during session startup. Invocations of the `Authenticate` method that use the Anybody authority shall always succeed. Values in the Challenge parameter of that `Authenticate` method shall be ignored.

The members of the Makers authority class permit the manufacturer of the TPer to open an authenticated session to the TPer. The MakerPuK (i.e., Manufacturer) authority only has the Manufacturer Public Key (not the private) and a Certificate attesting to this, which is signed by the Manufacturer.

The SID authority is used by the TPer owner to authenticate to the Admin SP and perform operations such as freezing or deleting SPs.

A copy of the SID is also present in each SP. This SID authority and credential provides the personalizing host with a default password authority that can be used to open sessions or verify physical presence. When an SP is issued or created, the value of the `Password` column of the `C_PIN` credential object referenced by the SID authority is the same as the value of the `Password` column of the `C_PIN` credential object referenced by the SID authority in the Admin SP. Modifications to the SID authority's referenced `C_PIN` credential object in some SP (even the Admin SP) do not affect any other SP.

The authorities TPerSign and TPerExch are references to the TPer's signing and exchange keys, and allow a host with knowledge of the TPer's credentials to open a secure session with an authenticated TPer. In the Admin SP `Authority` table, the `Credential` column contains the reference to locate the appropriate credential for use with this authority.

These TPerSign and TperExch authorities are present in the `Authority` table of each SP. The credentials to which these authorities contain references are represented by objects in the appropriate credential tables. The actual key values may be stored in only a single location, but the implementation shall maintain appropriate references to these credentials so that they are usable in each SP on the TPer.

In all SPs, the values TPerSign and TPerExchange in the Authority table's Operation column indicate that the signing or exchange operation is to be performed with the TPer credentials as referenced in the Admin SP's Authority table.

The AdminExch authority represents the initial credential value submitted during issuance. This is the authority that enables the host to open a secure, implicitly authenticated session to the host's SP and personalize that SP. In the case of the Authority table in the Admin SP, the Base Template Authority AdminExch shall be disabled. At issuance, prior to personalization, the AdminExch authority has a RespExch column value set to the AdminExch authority's UID.

5.3.4.1.3 Authority Operations

The Operation column of the Authority table identifies the authentication method for which an authority object shall be used.

The value of an Operation for a given authority shall match the purpose for which that authority is being used during session startup. For instance, an authority with an Operation value of Signing or None shall only be able to be successfully invoked during session startup as either a HostSigningAuthority or SPSigningAuthority.

The operation types and their requirements are as follows:

- **None –** This describes an authority that, while invoked during session startup, does not actually authenticate during session startup, but may be used to reference, for instance, a response signing or exchange authority. If invoked during session startup, an authority with this Operation column value shall be referenced as the HostSigningAuthority or SPSigningAuthority. Referencing an authority with this Operation column value as an exchange authority shall result in an error.
- **Password –** This describes an authority that shall be invoked and authenticated with its referenced C_PIN credential, either during session startup or using the Authenticate method. If invoked during session startup, this authority shall be referenced as the HostSigningAuthority or the SPSigningAuthority. Referencing an authority with this Operation column value in another authority parameter of the session startup methods shall result in an error.
- **Sign –** This describes an authority that shall be invoked and authenticated using a challenge and response with its referenced public key (RSA/EC) credential, either during session startup or using the Authenticate method. If invoked during session startup, an authority with this Operation column value shall be referenced as the HostSigningAuthority or as the SPSigningAuthority. Referencing an authority with this Operation column value in another authority parameter of the session startup methods shall result in an error. The Sign operation encompasses both Signing and Verification activities – the TPer shall perform the correct operation based on context.
- **Exchange –** This describes an authority that shall be invoked during session startup, and shall be referenced as the HostExchangeAuthority or the SPExchangeAuthority. Referencing an authority with this Operation column value in another authority parameter of the session startup methods shall result in an error. The credential referenced by this authority shall be used to encrypt session keys for transmission to the other party involved in the session. This authority shall not be able to be authenticated explicitly using the Authenticate method.
- **SymK –** This describes an authority that shall be invoked and authenticated using a challenge and response with its referenced symmetric key credential, either during session startup or using the Authenticate method. If invoked during session startup, an authority with this Operation column value shall be referenced as the HostSigningAuthority or the SPSigningAuthority. Referencing an authority with this Operation column value in another authority parameter of the session startup methods shall result in an error.
- **HMAC –** This describes an authority that shall be invoked and authenticated using a challenge and response with its referenced HMAC key credential and the referenced HMAC algorithm,

either during session startup or using the `Authenticate` method. The HMAC credential referenced by the authority using this operation identifies the hash algorithm used to generate the HMAC. If invoked during session startup, an authority with this `Operation` column value shall be referenced as the HostSigningAuthority or the SPSigningAuthority. Referencing an authority with this `Operation` column value in another authority parameter of the session startup methods shall result in an error.

- **TPerSign –** This describes the signing authority that represents the TPer, which enables the host to verify the TPer credentials. This authority shall be invoked and authenticated using a challenge and response with its referenced public key (RSA/EC) credential, either during session startup or using the `Authenticate` method. If invoked during session startup, an authority with this `Operation` column value shall be referenced as the HostSigningAuthority or as the SPSigningAuthority. Referencing an authority with this `Operation` column value in another authority parameter of the session startup methods shall result in an error. The Sign operation encompasses both Signing and Verification activities – the TPer shall perform the correct operation based on context. The TPer signing credential contains certificate chains that establish the validity of this authority.
- **TPerExchange –** This describes the exchange authority that represents the TPer. This authority enables the host to establish a secure session with an SP using the TPer's exchange authority. Referencing an authority with this `Operation` column value in another authority parameter of the session startup methods shall result in an error. The credential referenced by this authority shall be used to encrypt session keys for transmission to the other party involved in the session. This authority shall not be able to be authenticated explicitly using the `Authenticate` method. The TPer exchange credential contains certificate chains that establish the validity of this authority.

5.3.4.1.4 Session Startup

Session startup involves the exchange of either two or four methods between the host and the SP with which the host is attempting to start the session.

The properties of the session – i.e., whether secure messaging is required, the secure messaging type, and the type of message authentication – are controlled by values in the columns of authority objects referenced as parameters to the methods, and are determined independently for each communicator.

When the `StartSession` method is invoked, the authorities to be used for that session are referenced as parameters. The following identifies the order of authority precedence in the `StartSession` invocation. For the invoked host authorities, the following list defines the "Host Control Authority" that identifies the Host-to-SP session property requirements, including the secure messaging properties for communications from the host to the SP:

1. HostSigningAuthority
2. If no HostSigningAuthority is invoked, then the HostExchangeAuthority will be the "Host Control Authority".
3. If neither the HostSigningAuthority nor the HostExchangeAuthority invoked, then there will be no "Host Control Authority".

For SP response authorities referenced from the "Host Control Authority", the following list defines the "SP Control Authority" that identifies the SP-to-Host session property requirements, including the secure messaging properties for communications from the SP to the host:

1. SPSigningAuthority
2. If no SPSigningAuthority is referenced, then the SPExchangeAuthority will be the "SP Control Authority".
3. If neither the SPSigningAuthority nor the SPExchangeAuthority invoked, then there will be no "SP Control Authority".

If the StartSession method fails, the return result is formatted as a SyncSession method invocation from the TPer, using only the Host and SP parameters, with a non-Success status code.

If the StartTrustedSession method fails, the return result is formatted as a SyncTrustedSession method invocation from the TPer, using only the Host and SP parameters, with a non-Success status code.

5.3.4.1.5 Secure Messaging Control

As indicated in section 5.3.4.1.4, control of secure messaging for a session is determined independently for each communicator. The authorities invoked in the StartSession method determine the secure messaging types and algorithms requires, and, based on the authorities included in the session startup, the encrypting credential used to exchange session key(s) for secure messaging.

If the Host Signing Authority is invoked in StartSession, this authority determines if secure messaging is required on messages from the Host to the TPer, and of what type the secure messaging will be. In this circumstance, the Host Signing Authority is the "Host Control Authority" for messaging from the Host to the TPer.

If the Host Signing Authority is not present, and the Host Exchange Authority is present, the Host Exchange Authority determines if secure messaging is required on messages from the Host to the TPer, and of what type the secure messaging will be. In this circumstance, the Host Exchange Authority is the "Host Control Authority" for messaging from the Host to the TPer.

If the Host Signing Authority is the "Host Control Authority", its Response (SP) Signing and Response (SP) Exchange Authorities determine the authorities used to represent the SP. If the Host Signing Authority is not invoked in StartSession, and the Host Exchange Authority is, it is the Host Exchange Authority that is the "Host Control Authority" and its Response Signing and Response Exchange Authorities determine the authorities used to represent the SP.

If the Response (SP) Signing Authority is linked from the "Host Control Authority", it is considered the "SP Control Authority", and determines if secure messaging is required on messages from the TPer to the Host, and of what type the secure messaging will be. If the Response (SP) Signing Authority is not linked from the "SP Control Authority" and the Response (SP) Exchange Authority is, then it is the Response (SP) Exchange Authority that serves as the "SP Control Authority" and determines if secure messaging is required on messages from the TPer to the Host, and of what type the secure messaging will be.

If the value of a "Control Authority's" Secure column is 0, then secure messaging shall not be permitted for messaging from that communicator for the session to start successfully – all messaging exchanges for sessions controlled by that authority shall be in plaintext.

5.3.4.1.6 Hashing and Signing Method Parameters

If the Host Signing Authority is the "Host Control Authority", then its Authority.HashAndSign value identifies whether or not it is required that the parameters of the Host-to-TPer session startup methods (StartSession/StartTrustedSession) be hashed. If Authority.HashAndSign=T, then the parameters of these methods are hashed and signed using Authority.Credential and the hash protocol identified in that credential.

If the Host Exchange Authority is the "Host Control Authority", then its Authority.HashAndSign value identifies whether or not it is required that the parameters of the Host-to-TPer session startup methods (StartSession/StartTrustedSession) be hashed and signed. If Authority.HashAndSign=T, then the parameters of these methods are hashed and signed using Authority.Credential and the hash protocol identified in that credential.

If the Response (SP) Signing Authority is the "SP Control Authority", then its Authority.HashAndSign value identifies whether or not it is required that the parameters of the TPer-to-Host session startup methods (SyncSession/SyncTrustedSession) be hashed and signed. If Authority.HashAndSign=T,

then the parameters of these methods are hashed and signed using `Authority.Credential` and the hash protocol identified in that credential.

If the Response (SP) Exchange Authority is the "SP Control Authority", then its `Authority.HashAndSign` value identifies whether or not it is required that the parameters of the TPer-to-Host session startup methods (`StartSession/StartTrustedSession`) be hashed and signed. If `Authority.HashAndSign=T`, then the parameters of these methods are hashed and signed using `Authority.Credential` and the hash protocol identified in that credential.

5.3.4.1.7 Session Key Exchange

If the Host Signing Authority requires secure messaging, session keys are encrypted with the SP Exchange Authority's symmetric key or public key. If there is no SP Exchange Authority, or if there is an SP Exchange Authority but it does not reference a credential, then session keys are encrypted with the Host Exchange Authority's symmetric key. If there is also no Host Exchange Authority, or if there is a Host Exchange Authority but it does not reference an appropriate credential, an error shall be returned.

If the SP Signing Authority requires secure messaging, session keys are encrypted with the Host Exchange Authority's symmetric key or public key. If there is no Host Exchange Authority, or if there is a Host Exchange Authority but it does not reference a credential, then session keys are encrypted with the SP Exchange Authority's symmetric key. If there is also no SP Exchange Authority, or if there is an SP Exchange Authority but it does not reference an appropriate credential, an error shall be returned.

If there is no Host Signing Authority, the Host Exchange Authority may require secure messaging. If so, session keys are encrypted with the SP Exchange Authority's symmetric or public key. If there is no SP Exchange Authority, or if there is an SP Exchange Authority but it does not reference a credential, then session keys are encrypted with the Host Exchange Authority's symmetric key. If the Host Exchange Authority does not reference an appropriate credential, an error shall be returned.

If there is no SP Signing Authority, the SP Exchange Authority may require secure messaging. If so, session keys are encrypted with the Host Exchange Authority's symmetric key or public key. If there is no Host Exchange Authority, or if there is a Host Exchange Authority but it does not reference a credential, then session keys are encrypted with the SP Exchange Authority's symmetric key. If the SP Exchange Authority does not reference an appropriate credential, an error shall be returned.

When a key is required for integrity checking, that key shall always be exchanged as the HostIntegritySessionKey or SPIntegritySessionKey parameters in the `StartTrustedSession` or `SyncTrustedSession` method invocations. These keys are not used in the CCM and GCM authenticated encryption modes. When a key is used for message encryption, that key is exchanged as the HostEncryptSessionKey or SPEncryptSessionKey, even for authenticated encryption modes.

For secure messaging modes that use HMAC for message integrity, the following list identifies the size of the key that shall be exchanged as the integrity session key for the appropriate HMAC usage.

- For HMAC using SHA 256: The integrity session key shall be 256 bits.
- For HMAC using SHA 384: The integrity session key shall be 384 bits.
- For HMAC using SHA 512: The integrity session key shall be 512 bits.

For integrity algorithms that utilize public key cryptography, the Host uses the private key corresponding to the chained down certificate supplied in the `StartSession` algorithm to sign the hash of the message. The hash protocol for this operation is identified in the host control authority's credential. The SP uses the chained down certificate supplied in the `SyncSession` algorithm to sign the hash of the message. The hash protocol for this operation is identified in the SP control authority's credential.

5.3.4.1.8 Session Startup Authorities

If the HostSigningAuthority is specified in the `StartSession` invocation, and that authority has an `Operation` column value of Signing, SymK, or HMAC, then the SP shall respond to the `StartSession` invocation with a `SyncSession` invocation that contains the SPChallenge parameter, which holds a 32-byte nonce. The host shall then sign the SPChallenge nonce using the HostSigningAuthority's credential as appropriate, and then submit the signed challenge back to the SP in a `StartTrustedSession` call.

If the HostSigningAuthority is specified, and that authority has an `Operation` column value of Password, then the credential referenced by the authority shall be a `C_PIN` object for session startup to resolve properly. The host shall send the value of the PIN (in the clear) via the HostChallenge parameter in the `StartSession` method invocation.

If the SPSigningAuthority is referenced, and that authority has an `Operation` column value of Signing, SymK, or HMAC, then the host shall include in its `StartSession` method invocation the HostChallenge parameter, which holds a 32-byte nonce. The SP shall sign the HostChallenge nonce using the SPSigningAuthority's credential as appropriate, and submit the signed challenge back to the host in its `SyncTrustedSession` method.

If the SPSigningAuthority is referenced, and that authority has an `Operation` column value of Password, then the credential referenced by that authority shall be a `C_PIN` object for session startup to resolve properly. The SP shall send the value of the PIN (in the clear) via the SPChallenge parameter of the `SyncSession` method.

When session startup successfully completes, all authorities invoked during the session startup process shall be considered authenticated.

5.3.4.1.9 EC-MQV Session Startup

It is possible to use the session startup method exchanges to start a session using EC-MQV. In order to do so, it is necessary to follow the following protocol (See Figure 19 for more information):

- The host ECMQV ephemeral public key is conveyed in HostChallenge.
- The host ECMQV static public key is conveyed in HostExchangeCert
- The SP ECMQV ephemeral public key is conveyed in SPChallenge
- The SP ECMQV static public key is conveyed in SPExchangeCert
- The Full ECMQV, C(2,2,ECC MQV) scheme of NIST SP 800-56A, Section 6.1.1.4 is used.
- The key derivation function (KDF) is Concatenation KDF as defined in Section 5.8.1 of NIST SP 800-56A. The AlgorithmID is the ASCII encoding of the string "TCG Storage ECMQV". The PartyUInfo is the uinteger Host value and the PartyVInfo is the uinteger SP value. Supplementary fields are not used (that is, they are empty). The hash function used in the KDF is SHA-1 if the elliptic curve group is defined over a 163-bit or 192-bit field, is SHA-256 if the elliptic curve is defined over a 224-bit, 256-bit, 233-bit or 283-bit field, and is SHA-384 if the elliptic curve is defined over a 384-bit field.
- The C(2,2) "Bilateral Key Confirmation" as defined in Section 8.4.1 of NIST SP 800-56A is used (see below).
- The value MacTagU is conveyed in HostResponse
- The value MacTagV is conveyed in SPResponse

5.3.4.1.10 EC-DH Session Startup

It is possible to use the session startup method exchanges to start a session using EC-DH. In order to do so, it is necessary to follow the following protocol (See Figure 20 :

- The host ECDH static public key is conveyed in HostExchangeCert
- The host nonce (the value $Nonce_U$ used in key derivation and key confirmation) is conveyed in HostChallenge
- The SP ECDH static public key is conveyed in SPExchangeCert
- The SP nonce (the value $Nonce_V$ used in just key confirmation) is conveyed in the SPChallenge
- The Cofactor Static Unified Model C(0,2,ECC CDH) scheme of NIST SP 800-56A, Section 6.3.2 is used.
- The key derivation function (KDF) is Concatenation KDF as defined in Section 5.8.1 of NIST SP 800-56A. The AlgorithmID is the ASCII encoding of the string “TCG Storage ECDH”. The PartyUInfo is the uinteger Host value and the PartyVInfo is the uinteger SP value. Supplementary fields are not used (that is, they are empty). The hash function used in the KDF is SHA-1 if the elliptic curve group is defined over a 163-bit or 192-bit field, is SHA-256 if the elliptic curve is defined over a 224-bit, 256-bit, 233-bit or 283-bit field, and is SHA-384 if the elliptic curve is defined over a 384-bit field.
- The C(0,2) “Bilateral Key Confirmation” as defined in Section 8.4.8 of NIST SP 800-56A is used (see below).
- The value MacTagU is conveyed in HostResponse
- The value MacTagV is conveyed in SPResponse

5.3.4.1.11 Certificate Presentation

If an authority has a value of True in its `PresentCertificate` column; the authority references a public key credential; and that credential references a certificate, then a certificate chain must be presented when that authority is referenced as an SP authority.

For details on certificate contents/formatting, see the TCG Storage Certificates Specification.

5.3.4.1.12 Explicit Authentication with the Authenticate Method

In addition to authentication of authorities that participate in a successful startup of a session, authentication of an authority may also be achieved through successful invocation of the `Authenticate` method.

If the invoked authority requires password authentication (the value of the `Operation` column of the invoked authority is “Password”), one call to `Authenticate` is made and the Challenge parameter is the password. The response is either True or False – True if authentication was successful and False if authentication was unsuccessful.

If the authority requires challenge and response (the value of the `Operation` column of the authority is “Sign”, “SymK”, or “HMAC”), the host must invoke the `Authenticate` method twice. In the first invocation the method parameter list shall be empty. In the second invocation, the Authority parameter is the UID of the authority that the host is authenticating and the Challenge parameter is the response to the SP’s challenge. The Success response is returned to the second invocation– this will be either True if authentication was successful or False if authentication was unsuccessful.

If an attempt is made to invoke the `Authenticate` method on an authority that requires secure messaging, and the required secure messaging parameters as defined in the `Secure` column of the `Authority` table are not currently fulfilled, then the `Authenticate` method invocation shall fail. If an

attempt is made to invoke the Authenticate method on an authority that requires no secure messaging, and secure messaging is in operation, then the Authenticate method invocation shall fail.

When the Authenticate method invocation protocol requires the host to invoke the Authenticate method twice, the second Authenticate method may be sent anytime during the session (boundaries of nested transactions shall need to be respected). If the TPer receives any Authenticate method after the first Authenticate method has been invoked, the TPer shall attempt to resolve that authentication attempt, which shall fail if the second Authenticate does not contain the appropriate response parameters.

5.3.4.2 Table Management

5.3.4.2.1 Creating Tables

New tables are created via successful invocation of the CreateTable method. The Name-CommonName combination of the table created shall be unique in the Table table.

When a new table is created using the CreateTable method, the columns for the table are specified in the Columns parameter. The type of this parameter is a typeOr (for more information on types, see the format specification in section 5.1.1).

- The first option of this typeOr represents a list of column names and the uid of the type (from the Type table) to be associated with that column. This is to be used if the table does not have indexed columns.
- The second option of this typeOr represents a struct made up of two lists. The first list is made up of a list of column names and the uid of the type (from the Type table) to be associated with that column. The first list represents the set of columns in the table the combination of which is required to be unique (these are the indexed columns). The second list represents the set of columns in the table that are not part of the index of that table, and is a list of the column names and the uid of the type to be associated with that column.

The mechanism by which allocation of rows to a table is accomplished is implementation specific. A manufacturer may choose to allocate rows statically (create all rows at table creation) or dynamically (at each CreateRow method invocation).

The total number of rows that may be created for a table based on existing conditions shall be obtainable using the GetFreeRows method.

The CreateTable method uses the MinSize parameter to define the initial number of rows that shall be allocated for the new table. The created table shall always be able to have CreateRow invoked on it that many times. If the MinSize is too large (requests more rows than may be allocated for that table), the CreateTable method invocation shall result in an error.

MinSize is recorded in the MinSize column of the Table table. The MinSize column in the Table table may be changed using a Set method invocation. Access control requirements shall be fulfilled as normal. The TPer will return an error if an attempt is made to set a lower value than is recorded in the MinSize column. The TPer may reject the request and return an error.

The actual number of rows that have been created for a table are reflected in the value of the Table table's Rows column.

The optional MaxSize parameter defines the maximum number of rows that can be created for the table. Note that this is a host-supplied number, and that there are cases in which the created table shall not be permitted by the system to have MaxSize rows. However, the TPer shall guarantee that the table never has more than MaxSize rows.

The MaxSize parameter value is recorded in the MaxSize column of the Table table. Access control requirements shall be fulfilled as normal to permit this value to be changed, but the TPer may prevent the value from being changed – in such a case the TPer will return an error.

The optional HintSize is a number of rows larger than MinSize that is requested for the created table. The number of possible rows defined in HintSize is not required to be the number of possible rows actually supplied. It is a host-specified number of rows that the host suggests should be supplied by the system for that table, if sufficient amount of row space is available. This allows the TPer to optimize the allocation of rows for the table.

When a new table is created, in addition to space being allocated for that table, other side effects occur as well:

- A row in the `Table` table shall be created. This row is a table descriptor object that stores metadata about the newly created table, including the name of the table, the number of rows allocated for the table, and the number of free rows in the table. The table descriptor object also stores the type of the table (bytes, array, or object), as well as a reference to the `Column` table row of the table's first column. Tables and their `Table` table entries are differentiated from each other by different UID composition. A table is referred to by a UID derived from that of the associated `Table` table entry (see section 3.2.5.3).
- Rows in the `Column` table shall be created for each of the columns in the new table. Each row in the `Column` table stores metadata about a column in the newly created table, including the column's name and data type, whether the column value is part of the index for the table (requires uniqueness across the table), and whether the column is subject to transactional rollback, as well as the uidref to the row in the `Column` table that stores metadata about the next column in the created table.
- Rows in the `Method` table shall be created for each of the methods available for both the newly created table and its associated `Table` table row.
- Rows in the `ACE` table may be created to limit access control on certain portions of the table or its associated `Table` table row.

If all associated rows in all associated tables cannot be created, then the invocation of the `CreateTable` method shall fail.

5.3.4.2.2 Retrieving Table Data and Metadata

Table information can be classified in two ways:

- Table Data – this refers to the data stored in the cells of the table.
- Table Metadata – this refers to data that is stored about the table.

Table data is stored in the table itself, while table metadata is stored in other tables, called Metadata Tables. Examples of these tables are the `Table` table and the `Column` table.

Rows in the `Table` table are table descriptor objects. Since each table descriptor is an object, each row in that table, like any other object, has its own methods that may be used to retrieve the data stored in that object. So, upon creation of a table, the control authority for a session is set in the access control associations for the `Get` method that enable the host to retrieve that table descriptor object's data after authenticating that authority.

Table data, the data stored in table cells, may be retrievable through successful invocation of the `Get` method on that table. When the `Get` method is invoked on a table or object, only the values that are readable based on currently authenticated authorities and their associated ACE restrictions for the method shall be returned.

Cell values that may have been requested but are not permitted to be read by the currently authenticated authorities are not returned. Since the return value of the method for non-byte tables is a list of name-value pairs, cells to which the host invoking the `Get` method does not have access are omitted from the return result. If a column is known to exist but not returned with a value, then the host can discern that it did not have permission to invoke `Get` on that cell.

5.3.4.2.3 Creating Table Rows and Objects

Tables may be modified in the following ways:

- New rows may be created
- Existing rows may be deleted
- Cell values may be changed

In most cases, new rows shall be added to a table through successful invocation of the `CreateRow` method on that table. Exceptions to this, where rows are added by methods other than `CreateRow`, are identified in table descriptions and in the Template-specific Default Access Control Settings sections of this specification.

Successful invocation of the `CreateRow` method creates a row in the invoking table where the column values for that row are the values passed as parameters to the method invocation. For columns that did not have values defined in the `CreateRow` invocation, default values are assigned as described in 5.3.2.5.

When invoking the `CreateRow` method on a table that requires certain column values to be unique, an attempt to create a row with parameterized indexed values equivalent to the values in the indexed columns of some row of that table shall fail unless access control requirements that permit the `Delete` or `DeleteRow` method to be invoked on that table, encompassing the entirety of the affected row (based on parameterized column values), have also been fulfilled. If access control is fulfilled, the data parameterized in the `CreateRow` method overwrites the data in corresponding columns of that row.

When a row in an object table is created, a number of `Method` table rows are created that correspond to the default methods permitted for the created object. ACLs are set on those methods, and in the meta-ACL methods associated with those methods, as follows:

- Using the HostSigningAuthority from the `StartSession` method, if provided.
- Otherwise, using the HostExchangeAuthority from the `StartSession` method, if provided.
- Otherwise, using the Anybody Authority.

5.3.4.2.4 Deleting Table Rows and Objects

Rows may be deleted from a table in two ways:

- Successful invocation of the `DeleteRow` method on the table
- Successful invocation of the `Delete` object method on an object

Deleting objects may have side effects. For instance, invoking the `Delete` method on a `Table` table row has the side effect of deleting the table with which that table descriptor object is associated. Side effects that occur upon deletion of objects are documented where appropriate.

Deleting objects shall also cause all `Method` table rows associated with that object to be deleted.

When an object is deleted by a successful invocation of the `DeleteRow` method, the side effects of the method are the same as if the object had been deleted via invocation of the `Delete` method (see 5.3.3.3).

5.3.4.2.5 Deleting Tables

As indicated in Section 5.3.4.2.4, a table shall be deleted by successful invocation of the `Delete` method or the `DeleteRow` method on the table descriptor object (`Table` table row) associated with the table to be deleted.

When the method resolves, the following occurs:

- The table descriptor object associated with the table is deleted.
- The table itself is deleted.
- All associated `Method` table rows are deleted. This includes methods associated with both the table itself as well as the table descriptor object.
- All associated `Column` table rows shall be deleted.

Due to their reusability, ACEs that may have been created when a table was created shall not be deleted when that table is deleted, as the ACE may still be in use by another table. ACEs created as a side effect of creating a table or object shall be deleted only by direct host action.

5.3.4.2.6 Modifying Tables

In most cases, modifications to tables can be accomplished using the `Set` method (access control permitting). Other cases that allow modification of tables without use of the `Set` method are noted in the appropriate document section.

Unlike with the `Get` method, when the `Set` method is invoked on a table but access control does not permit some cell to be changed that the `Set` method invocation is attempting to change, the entire method fails and returns an error. All changes parameterized in the `Set` method shall be made for the method to resolve successfully.

5.3.4.3 Access Control

Access control describes the system used to prevent modification of an SP's contents by a host that does not have proper authorization to make those modifications.

The `Method` table stores access control associations between methods and the tables, objects, or SP upon which those methods may operate. Each row is an access control association made up of a reference to the method and a reference to the object/table/SP; an Access Control List (ACL); meta-ACLs; and columns that store the logging settings for the access control association and its associated meta-ACL methods.

Each `InvokingID/MethodID` combination in the `Method` table shall be unique within the table.

Each ACL column shall hold a limited number of ACEs. The actual number that each ACL column may store is SSC-dependent.

The default ACEs provided by the Base Template appear in Table 74. Their use is defined in Section 5.3.5 and other Template Life Cycle sections in this document.

Table 74 Base Template Default ACEs

UID	Name	BooleanExpr	RowStart	RowEnd	ColStart	ColEnd
00 00 00 08 00 00 00 01	Anybody	Anybody				
00 00 00 08 00 00 00 02	Admins	Admins				
00 00 00 08 00 00 00 03	Makers	Makers				

UID	Name	BooleanExpr	RowStart	RowEnd	ColStart	ColEnd
00 00 00 08 00 00 00 04	PostIssuanceAdmins	Admins	0	0		
00 00 00 08 00 00 00 05	SPInfo_1	Admins	1	1	Enabled	Enabled
00 00 00 08 00 00 00 06	Table_Size	Admins			MinSize	MaxSize
00 00 00 08 00 00 00 07	Method_1	Admins			Log	Log
00 00 00 08 00 00 00 08	Method_2	Admins			AddACELog	LogTo

The `ACE` table for all SPs is defined at Issuance shall be composed of at least the ACE table entries in Table 74. These entries define Boolean combination of Authorities from the `Authority` table (displayed in the ACE column). These are entries for the Base Template, and so exist for all SPs.

Of importance is the notation 0-0 for the Row restriction on the PostIssuanceAdmins ACE. This means starting at the row of a table after the last row established at the point of Issuance, which is defined as the point where the IssueSP command returns success. It ends at the last row of the table. The reserved row number 0 is, in the context of specifying an ACE row restriction, both the beginning of PostIssuance rows and the end of all rows in a particular table.

5.3.4.3.1 Meta-ACLs

The ability to add ACEs to or delete ACEs from ACLs provides additional granularity of access control, as the authorization required to add or remove ACEs from ACLs may be different from the authorization required to invoke the method described in the access control association.

The methods that perform the operations that add or remove ACEs from ACLs, or retrieve the list of ACEs from an ACL, are `AddACE`, `RemoveACE`, and `GetACL`. Additionally, access control associations may be removed from the `Method` table through successful invocation of the `DeleteMethod` method. These four methods together are the meta-ACL methods.

Each access control association stores ACLs that govern the authorization requirements for these methods on that access control association. A separate column in the `Method` table exists for each of the meta-ACL methods to store the ACL for that meta-ACL method. A separate column in the `Method` table also exists for each of the meta-ACL methods to store the logging settings for that meta-ACL method.

In order to add an ACE to an ACL, the ACL for the `AddACE` method associated with that access control association (stored in the `AddACEACL` column) must be satisfied. `RemoveACE`, `GetACL`, and `DeleteMethod` function in a similar way, and would have to fulfill the ACL stored in the `RemoveACEACL`, `GetACLACL`, and `DeleteMethodACL` column respectively.

5.3.4.4 Default Logging Settings

The default logging settings associated with the Template methods assume that the Log Template has been issued with the SP. Otherwise, these values should be disregarded, as values of log control columns shall be ignored if the Log Template has not been issued with an SP.

- Session startup logging (controlled in the `Authority` table) and logging invocation of the `Authenticate` method shall have default settings of LogAlways.
- The following method invocations shall by default log as LogAlways:

- AddACE
- RemoveACE
- DeleteMethod
- Delete
- CreateTable
- CreateRow
- DeleteRow

- Invocation of the following method invocations shall by default log as LogFailure:
  - DeleteSP
  - Set
  - GenKey
- All other methods described in the Base Template shall be default log as LogNever.

5.3.5 Life Cycle

5.3.5.1 Base Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Base Template has the following characteristics based on the current life cycle state of that SP:

- **Issued –** At Issuance the SP will have the default Base Template-related access control settings as described in the following sections.
- **Disabled –** A Base Template-enabled SP that is in the Disabled state shall not be able to perform any user-invoked SP operations enabled, with the exceptions noted in section 4.4.3. These exceptions include invocation of the `Authenticate` method, the `DeleteSP` method, and the `Set` method used to re-enable the SP. Session Manager protocol layer methods invoked to the disabled SP shall operate as normal.
- **Frozen –** A Base Template-enabled SP that is in the Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.4.
- **Issued-Disabled-Frozen** – A Base Template-enabled SP that is in the Issued-Disabled-Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.5.

5.3.5.2 Initial Access Control Settings

The following sections enumerate the initial required access control settings for the object/method, table/method, and SP/method combinations provided to an SP by the Base Template. These access controls represent the pre-personalization settings of the Base Template-related table/method and SP/method combinations, i.e. those when the SP initially enters the Issued state.

In the descriptive tables in this section, “None” indicates that the relevant ACL column of the `Method` table has a Null uidref (zeroes). This indicates that access control to perform that action cannot be satisfied.

Some methods do not appear in the descriptive tables in this section for some Template tables or objects. This indicates that the method shall not be able to be invoked on that table or object, and there shall be no row in the `Method` table representing that access control association.

5.3.5.2.1 ACEs

ACEs defined as defaults for the Base Template, which are defaults for all SPs, are defined in Table 74.

5.3.5.2.2 SP Methods Default Access Control Settings

Table 75 Base Template SP Method Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
DeleteSP	Admins	Admins	Admins	Admins	Admins
CreateTable	Admins	Admins	Admins	Admins	Admins
GetFreeSpace	Admins	Admins	Admins	Admins	Admins
Authenticate	Admins	Admins	Admins	Admins	Admins

5.3.5.2.3 SPInfo Table Default Access Control Settings

Table 76 SPInfo Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	SPInfo_1	None	Admins	Admins	Admins

5.3.5.2.4 SPTemplates Table Default Access Control Settings

Table 77 SPTemplates Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins

5.3.5.2.5 Table Table and Table Descriptor Object Default Access Control Settings

Table 78 Table Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
DeleteRow	PI_Admins	None	Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	Table_Size	None	Admins	Admins	Admins

Table 79 Table Descriptor Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
GetFreeRows	Self	Self	Self	Self	Self
Set	Table_Size	None	Self	Self	Self

5.3.5.2.6 Column Table Default Access Control Settings

Table 80 Column Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins

5.3.5.2.7 MethodID Table and MethodID Object Default Access Control Settings

Table 81 MethodID Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins

5.3.5.2.8 Method Table and Method Object Default Access Control Settings

Table 82 Method Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	Method_1, Method_2	None	Admins	Admins	Admins

5.3.5.2.9 Type Table and Type Object Default Access Control Settings

Table 83 Type Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins

Table 84 Type Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self

5.3.5.2.10 ACE Table and ACE Object Default Access Control Settings

Table 85 ACE Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Set	PI_Admins	None	PI_Admins	Admins	Admins

Table 86 ACE Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

5.3.5.2.11 Authority Table and Authority Objects Default Access Control Settings

Table 87 Authority Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	PI_Admins	None	PI_Admins	Admins	Admins

Table 88 Authority Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

5.3.5.2.12 Certificates Table and Certificates Object Default Access Control Settings

Table 89 Certificates Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	PI_Admins	None	PI_Admins	Admins	Admins

Table 90 Certificates Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

5.3.5.2.13 C_PIN Table and C_PIN Object Default Access Control Settings

Table 91 C_PIN Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	PI_Admins	None	PI_Admins	Admins	Admins

Table 92 C_PIN Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
GenKey	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

5.3.5.2.14 C_RSA_* Tables and Objects Default Access Control Settings

Table 93 C_RSA_* Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	PI_Admins	None	PI_Admins	Admins	Admins

Table 94 C_RSA_* Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
GenKey	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

5.3.5.2.15 C_AES_* Tables and Objects Default Access Control Settings

Table 95 C_AES_* Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	PI_Admins	None	PI_Admins	Admins	Admins

Table 96 C_AES_* Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
GenKey	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

5.3.5.2.16 C_EC_* Tables and Objects Default Access Control Settings

Table 97 C_EC_* Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	PI_Admins	None	PI_Admins	Admins	Admins

Table 98 C_EC_* Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
GenKey	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

5.3.5.2.17 C_HMAC_* Tables and Objects Default Access Control Settings

Table 99 C_HMAC_* Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	PI_Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	PI_Admins	None	PI_Admins	Admins	Admins

Table 100 C_HMAC_* Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
GenKey	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

5.3.6 Examples

This section details pseudo-code examples utilizing Base Template methods and tables, as well as examples using the Session Manager protocol layer methods.

5.3.6.1 Session Startup Examples

The session startup examples in this section use the example Authority Table 101. References to objects (uidrefs) in the following examples are shorthanded by providing a descriptive name to represent that credential (for instance, Authority object uids other than for the Anybody authority are named AuthUID1-AuthUID5. Credential objects uidrefs are represented by the credential type, and a number (PINUID1 represents the first `C_PIN` object referenced in the example).

Table 101 Authority Table (Example) – Session Startup

UID	Name	IsClass	Enabled	Secure	Operation	Credential	ResponseSign	ResponseExch
AnybodyUID	Anybody	F	T	0	0	0	0	0
AuthUID1	NAME1	F	T	0	1	PINUID1	0	0
AuthUID2	NAME2	F	T	C_AES_128	3	RSA1024UID1	AuthUID4	AuthUID5
AuthUID3	NAME3	F	T	0	2	AES128UID1	0	0
AuthUID4	NAME4	F	T	0	3	RSA1024UID2	0	0
AuthUID5	NAME5	F	T	0	2	AES128UID2	0	0

5.3.6.1.1 Anybody Authority Session Example

```
SMUID.StartSession [HostSessionID, SPID, 0]
=>
SMUID.SyncSession [HostSessionID, SPSessionID]
```

5.3.6.1.2 PIN Authority Session Example

Table 102 C_PIN Table (Example) – Session Startup

UID	Name	PIN
PINUID1	NAME1	ABC123

```
SMUID.StartSession [HostSessionID, SPID, 0, HostChallenge = 'ABC123',
HostSigningAuthority = AuthUID1]
=>
SMUID.SyncSession [HostSessionID, SPSessionID]
```

5.3.6.1.3 Authenticated and Secure Session Example

This example utilizes the example authorities detailed in Table 101.

```
SMUID.StartSession [HostSessionID, SPID, 0, HostChallenge = HostChallenge,
HostExchangeAuthority = AuthUID3, HostSigningAuthority = AuthUID2]
=>
SMUID.SyncSession [HostSessionID, SPSessionID, SPChallenge = SPChallenge]

SMUID.StartTrustedSession [HostSessionID, SPSessionID, HostResponse =
Signed<SPChallenge>, HostEncryptSessionKey = Encrypt<HostSessionKey>]
=>
```

```
SMUID.SyncTrustedSession [HostSessionID, SPSessionID, SPResponse =
Signed<HostChallenge>, SPEncryptSessionKey = Encrypt<SPSessionKey>]
```

5.3.6.2 CreateTable Example

Note that in this example, only columns relevant to the example are displayed in the sample table.

```
SPUID.CreateTable [‘DemoTable’, 1, ACEUID, []‘Name’, TypeUID2], [‘Value’, TypeUID3],
[‘State’, TypeUID4]], 10]
=>
[[DemoTableUID, 10]]
```

Table 103 Table Table (Example) – CreateTable

UID	Name	Kind	Column	Rows	RowsFree
DemoTableUID	DemoTable	1	RowNo1	10	10

Table 104 Column Table (Example) – CreateTable

Row	UID	Name	Type	Next
RowNo1	DemoColumn1	UID	TypeUID1	RowNo2
RowNo2	DemoColumn2	Name	TypeUID2	RowNo3
RowNo3	DemoColumn3	Value	TypeUID3	RowNo4
RowNo4	DemoColumn4	State	TypeUID4	0

Table 105 DemoTable Table (Example) – CreateTable

UID	Name	Value	State

5.3.6.3 CreateRow Example

```
DemoTableUID.CreateRow [[Name=‘NAME1’, Value=VALUE1, State=STATE1], [Name=‘NAME2’,
Value=VALUE2, State=STATE2], [Name=‘NAME3’, Value=VALUE3, State=STATE3],
[Name=‘NAME4’, Value=VALUE4, State=STATE4]]
=>
[{UID1, UID2, UID3, UID4}]
```

Table 106 Demo Table (Example) – CreateRow

UID	Name	Value	State
UID1	NAME1	VALUE1	STATE1
UID2	NAME2	VALUE2	STATE2
UID3	NAME3	VALUE3	STATE3
UID4	NAME4	VALUE4	STATE4

5.3.6.4 DeleteRow Example

```
DemoTableUID.DeleteRow [[UID1]]
=>
[1]
```

Table 107 Demo Table (Example) – DeleteRow

UID	Name	Value	State

UID	Name	Value	State
UID2	NAME2	VALUE2	STATE2
UID3	NAME3	VALUE3	STATE3
UID4	NAME4	VALUE4	STATE4

5.3.6.5 Delete Example

```
UID2.Delete []
=>
[1]
```

Table 108 Demo Table (Example) – Delete

UID	Name	Value	State
UID3	NAME3	VALUE3	STATE3
UID4	NAME4	VALUE4	STATE4

5.3.6.6 Get Examples

The examples in this section use Table 108

5.3.6.6.1 Get (Table Method) Example

```
DemoTableUID.Get [[startColumnName='Value', endColumnName='Value']]
=>
[[['Value'=VALUE3], ['Value'=VALUE4]]]
```

5.3.6.6.2 Get (Object Method) Example

```
UID3.Get [[startColumnName='Value', endColumnName='Value']]
=>
[[['Value'=VALUE3]]]
```

5.3.6.7 Set Examples

5.3.6.7.1 Set (Table Method) Example

```
DemoTableUID.Set [[startRow=UID3, endRow=UID3], [[Name=NAME1, Value=VALUE1]]]
=>
[1]
```

Table 109 Demo Table (Example) – Set

UID	Name	Value	State
UID3	NAME1	VALUE1	STATE3
UID4	NAME4	VALUE4	STATE4

5.3.6.7.2 Set (Object Method) Example

```
UID4.Set[[[Name=NAME2, Value=VALUE2]]]
=>
[1]
```

Table 110 Demo Table (Example) – Set

UID	Name	Value	State
UID3	NAME1	VALUE1	STATE3
UID4	NAME2	VALUE2	STATE4

5.3.6.8 Next Examples

Examples in this section refer to sample Table 110.

```
DemoTableUID.Next[]
=>
[[UID3, UID4]]

DemoTableUID.Next [Where = UID3]
=>
[[UID4]]
```

5.3.6.9 Authenticate Examples

Note that in the `Authenticate` method examples, only columns relevant to the example are displayed in the sample tables.

5.3.6.9.1 Authenticate Method Example – PIN

Table 111 Authority Table (Example) – Authenticate

UID	Name	IsClass	Enabled	Operation	Credential
AuthUID1	NAME1	F	T	1	PINUID1

Table 112 C_PIN Table (Example) – Authenticate

UID	Name	PIN
PINUID1	NAME1	ABC123

```
SPUID.Authenticate [AuthUID1, Challenge='ABC123']
=>
[1]
```

5.3.6.9.2 Authenticate Method Example – Challenge/Response

Table 113 Authority Table (Example) – Authenticate

UID	Name	IsClass	Enabled	Operation	Credential
AuthUID2	NAME1	F	T	3	RSA1024UID1

```
SPUID.Autheticate [AuthUID2]
=>
[ResponseBytes]

SPUID.Autheticate [AuthUID2, Challenge=Signed<ResponseBytes>]
=>
[1]
```

5.3.6.10 AddACE Example

Table 114 Method Table (Example) – AddACE

UID	MethodID	Type	ACL	AddACEACL	RemoveACEACL	DeleteMethodACL
UID	SetMethodUID	DemoTableUID	ACEUID	ACEUID1, ACEUID2	ACEUID3	ACEUID4

```
MethodTableUID.AddACE [DemoTableUID, SetMethodUID, ACEUID5]
=>
[1]
```

Table 115 Method Table (Example) – AddACE Result

UID	MethodID	Type	ACL	AddACEACL	RemoveACEACL	DeleteMethodACL
UID	SetMethodUID	DemoTableUID	ACEUID1, ACEUID5	ACEUID1, ACEUID2	ACEUID3	ACEUID4

5.3.6.11 RemoveACE Example

```
MethodTableUID.RemoveACE [DemoTableUID, SetMethodUID, ACEUID1]
=>
[1]
```

Table 116 Method Table (Example) – RemoveACE

UID	MethodID	Type	ACL	AddACEACL	RemoveACEACL	DeleteMethodACL
UID	SetMethodUID	DemoTableUID	ACEUID5	ACEUID1, ACEUID2	ACEUID3	ACEUID4

5.3.6.12 DeleteMethod Example

```
MethodTableUID.DeleteMethod [DemoTableUID, SetMethodUID]
=>
[1]
```

Table 117 Method Table (Example) – DeleteMethod

UID	MethodID	Type	ACL	AddACEACL	RemoveACEACL	DeleteMethodACL

5.3.6.13 Authority Table Example

Every SP that incorporates the Base Template shall have the authorities identified in Table 73. The specific values of the credentials may differ depending on the particular SP, and the capabilities of the TPer on which it exists. Table 118 provides some example customizations of other portions of the `Authority` table.

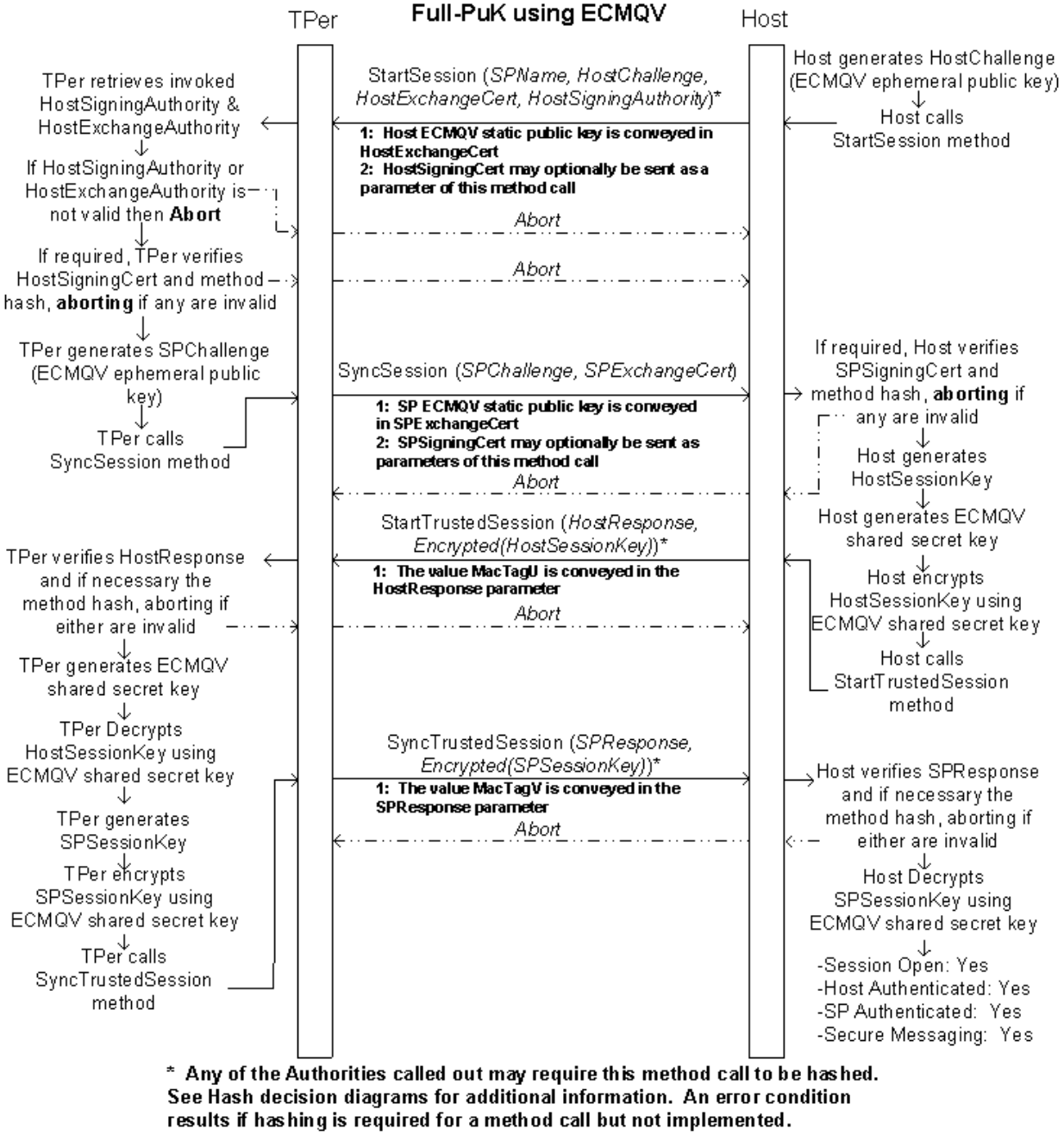
Table 118 Example Authority Table

Name	Common Name	IsClass	Class	Enabled	Secure	Operation	CredentialTable	Cre-dential	Resp Sign	Resp Exch	Log
Anybody	Anybody	False	None	True	NONE	NONE	0	0	0	0	LogNever
Admins	Admin	True	None	True	NONE	NONE	0	0	0	0	LogAlways
Makers	Maker	True	None	True	NONE	NONE	0	0	0	0	LogAlways
MakerSymK	Maker	False	Makers	True	C_AES_128	SymK	C_AES_128	1	0	0	LogNever
MakerPuK	Maker	False	Makers	True	NONE	Signing	C_RSA_2048	1	0	0	LogNever

Name	Common Name	IsClass	Class	Enabled	Secure	Operation	CredentialTable	Cre-dential	Resp Sign	Resp Exch	Log
SID	TPerOwner	False	None	True	NONE	Password	C_PIN	1	0	0	LogFail
TPerSign	TPerSign	False	None	True	NONE	TPerSign	C_RSA_1024	2	0	0	LogFail
TPerExch	TPerExch	False	None	True	NONE	TPerExchange	C_RSA_1024	3	0	0	LogFail
AdminExch	Admin	False	Admins	True	NONE	Exchange	C_AES_128	2	0	0	LogFail

5.3.6.14 Starting Sessions Using EC-MQV

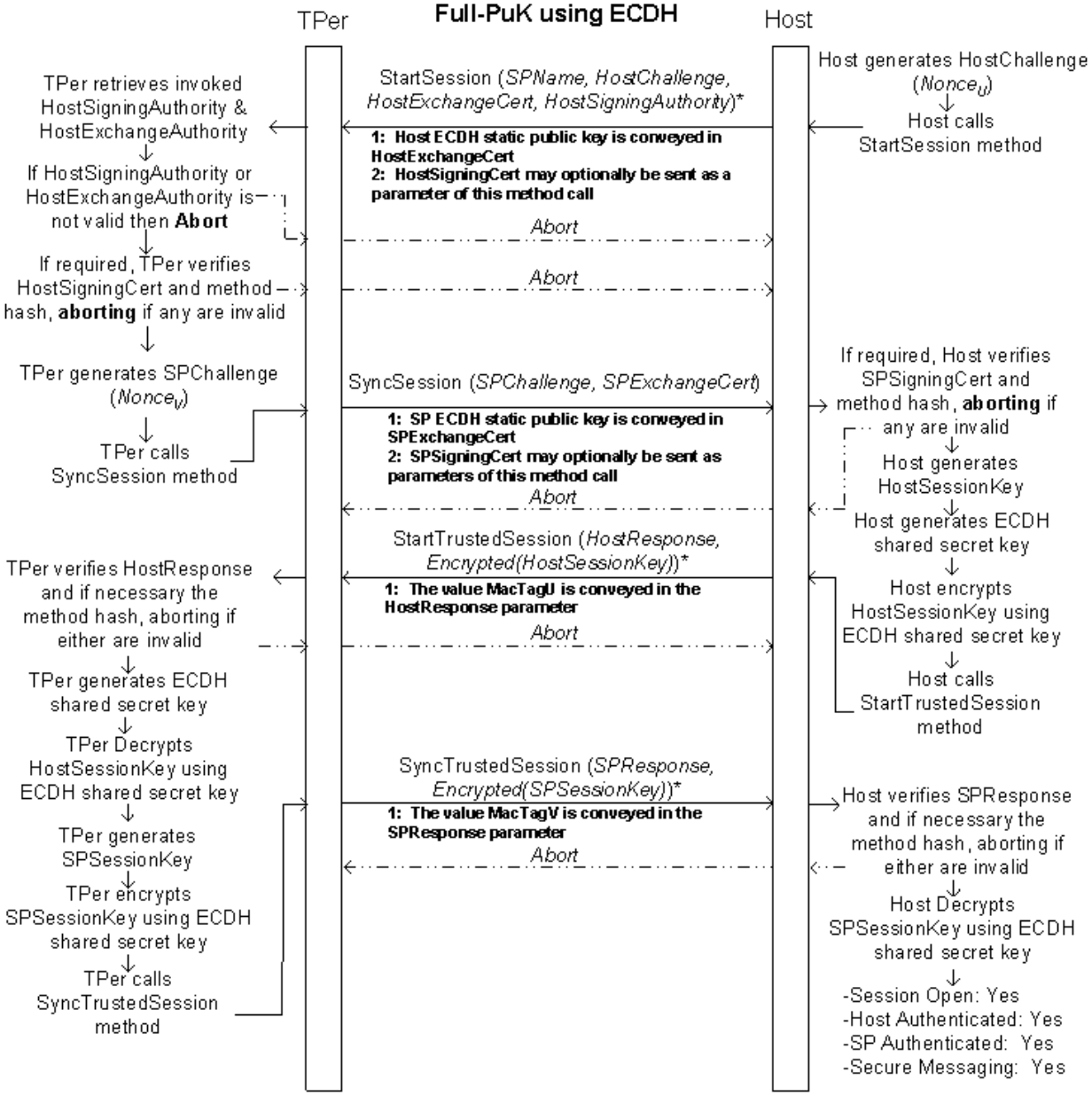
Figure 19 Starting Sessions Using EC-MQV



*** Any of the Authorities called out may require this method call to be hashed. See Hash decision diagrams for additional information. An error condition results if hashing is required for a method call but not implemented.**

5.3.6.15 Starting Sessions Using EC-DH

Figure 20 Starting Sessions Using EC-DH



*** Any of the Authorities called out may require this method call to be hashed. See Hash decision diagrams for additional information. An error condition results if hashing is required for a method call but not implemented.**

5.4 Admin Template

5.4.1 Overview

The purpose of the Admin Template is to provide to the Admin SP the capability to optionally Issue additional SPs and to maintain information about the TPer.

5.4.2 Data Structures

5.4.2.1 TPer Metadata Group - TPerInfo (Array Table)

The table in this section describes the metadata that the Admin SP stores about the TPer.

Table 119 TPerInfo Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	The UID of this row. (Read-only)
Bytes	uinteger_8	The size in bytes of the TPer's entire protected storage area. (Read-only)
GUDID	bytes_12	TPer's globally unique serial number - See Table 120 and accompanying text for GUDID content description (Read-only)
Generation	uinteger_4	Generation number of the volume. (Read-only)
FirmwareVersion	uinteger_4	Manufacturer-defined revision number of the TPer firmware. (Read-only)
ProtocolVersion	uinteger_4	Revision number of the interface messaging protocol, defined by the TCG Specification (Read-only)
SpaceForIssuance	uinteger_8	Amount of available bytes remaining for issuance. (Read-only)
SSC	name	Unique name of the SSC, as defined by TCG, that is supported by the TPer

The `TPerInfo` table contains exactly one row that is always readable by the Anybody authority.

5.4.2.2 TPer Metadata Group - Serial Number Contents

Table 120 GUDID Column Contents Description

Byte\Bit	7	6	5	4	3	2	1	0
0	0x02							
1	0x23							
2	0x00							
3	0x08							
4	NAA (0x05)				(MSB) IEEE COMPANY ID			
5	IEEE COMPANY ID							
6								
7	IEEE COMPANY ID (LSB)				(MSB) VENDOR-SPECIFIC IDENTIFIER			
8	VENDOR-SPECIFIC IDENTIFIER							
9								
10								
11	VENDOR-SPECIFIC IDENTIFIER (LSB)							

This structure meets the requirements of an identification descriptor as in SPC-3, and specifically conforms to the NAA IEEE Registered format defined in that document.

5.4.2.3 TPer Metadata Group - CryptoSuite (Array Table)

The table in this section describes the metadata the TPer keeps about itself

Table 121 CryptoSuite Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	The UID of this Crypto operation row. (Read-only)
CryptoCall	name	Name of the Crypto type (Read-only)
CryptoLen	uinteger_2	Key length for this CryptoCall (Read-only)
CryptoOp	name	Name of the Crypto operation being timed for this CryptoType/CryptoLen combination (i.e., KeyGen, Encrypt, Decrypt, Sign, Verify, Hash) (Read-only)
Special	boolean_def_false	Defines if special operating properties exist for this CryptoType/CryptoLen combination. Default value is “False” (Read-only)
Time	uinteger_4	Nominal Operation Time in Milliseconds (Read-only)
Variance	uinteger_4	Nominal Operation Time or Variance in Milliseconds, as applicable. (Read-only)

The times recorded in the `CryptoSuite` table must be average time based on 100 independent samples using randomly generated keys. The times may be taken when the TPer is otherwise idle and represent relative performance of the operations, not a guarantee of actual performance in the field.

Every type of crypto functionality present on the TPer shall have one row where the value of the `Special` column is False, and may have one or more rows where the value of the `Special` column is True. Rows where `Special`=True represent functionality that may be defined by special properties of the device such as hardware accelerators or pre-computed cache values (in the case, for example, of some key generation or random number provisioning).

Note: The rows in this table shall represent all of the crypto functionality on the TPer available to any/all newly issued SPs.

5.4.2.4 TPer Metadata Group – Properties (Byte Table)

The `Properties` table is a byte table that stores information about the communications properties of the TPer. The table is read-only, and its size is SSC-specific. The `Properties` table shall be sorted as the spec indicates to provide the required response to the `Properties` method.

For information on the `Properties` method and values required by the Core Specification to be stored in the `Properties` table, see Table 32.

Implementation-specific `Properties` may also be stored in the `Properties` table, but shall appear at the end of the table, after all of the spec required name-value pairs.

5.4.2.5 SPs on the TPer Group - Template (Object Table)

The table in this section describes the data that the Admin SP keeps about all of its Templates.

Table 122 Template Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier of this Template. (Read-only)
Name	Yes	name	Unique name of this Template. (Read-only)
RevisonNumber		uinteger_4	For Templates defined by the TCG Core Specification, this is the TCG Core Spec revision number of the Template (Read-only)
Instances		uinteger_2	Number of SPs on the TPer that are currently instantiated from this Template. Only deleting an SP can decrement this number. (Read-only)
MaxInstances		uinteger_2	Maximum number of SPs that may be instantiated from this Template at any one time. If 0, then there is no limit on number of instances. (Read-only)

The `Template` table has one row for each Template that may be issued by the TPer.

If the value of the `Instances` column is equal to the value of the `MaxInstances` column for a given Template, then attempts to issue additional SPs incorporating that Template shall result in an error.

5.4.2.6 SPs on the TPer Group - SP (Object Table)

The table in this section describes the data that the Admin SP keeps about all of the SPs on the TPer.

Table 123 SP Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier of this SP. (Read-only)
Name	Yes	name	Unique name of this SP. (Read-only)
ORG		Authority_ref	The Root Authority that authorized this SP (Read-only)
EffectiveAuth		bytes_32	The Chained Down Public Key of the Authority that actually issued this SP (Read-only)
DateofIssue		date	The date of Issuance (enabled by Clock Template) (Read-only)
Bytes		uinteger_8	Size of the SP. (Read-only)
LifeCycleState		life_cycle_state	Life Cycle state of this SP (Read-only)
Frozen		boolean_def_false	TPer Owner control over whether sessions may be opened on this SP. A value of True in this column indicates that attempts to open sessions to this SP fail. The default value of this column is False.

The Admin SP is always `UID=0x00 0x00 0x02 0x05 0x00 0x00 0x00 0x01` in the `SP` Table, and the values in this table are always readable by the Anybody authority.

NOTE: The `LifeCycleState` column cannot be written directly - the TPer changes it as appropriate.

5.4.3 Methods

5.4.3.1 IssueSP (SP Method)

This method is used to issue SPs on those TPers where the SPs are not fixed by the manufacturer.

```
IssueSP[
       SPName : name,
       Size : uinteger_4,
       Templates : template_list,
       AdminExch : exchange_key,
       Enabled : boolean ]
=>
[ UID : SP_ref,
Size : uinteger_4 ]
```

The IssueSP method creates a new SP of the specified name from the given **Templates**. The AdminExch authority in the new SP will be given the key defined in the **AdminExch** parameter in its newly created credential.

The **Size** is the size in 512 byte blocks that is requested for this SP. The returned Size is the size actually allocated. The Size returned shall be equal to or greater than the Size requested. If the TPer cannot allocate the requested Size, the IssueSP method invocation will result in an error.

Issuance always assumes the Base Template, but it is not an error to list it. The list of Templates may include any Template except the Template named "Admin," though the Templates available for use are restricted by the MaxInstances allowed for each Template. The Templates parameter is a list of UIDs of templates to be included in the issued SP. The UIDs are those of the templates as recorded for each template in the Admin SP's Template table.

The methods and tables from the Templates requested become part of the issued SP.

ORGs that are permitted to only issue certain Templates into new SPs are controlled by attributes in the ORG's certificate. This is defined in the TCG Storage Certificates specification. Reference that document for details.

5.4.3.1.1 Fails

- If there is already an SP of the same name.
- If the maximum number of SPs permitted for this template already exist.
- If there's not enough free space for the new SP in the TPer.

5.4.4 Descriptions

There shall be exactly one Admin SP on every TPer that has SPs. The Admin SP shall not be able to be disabled or deleted.

For TPers that have SPs, when a TPer is shipped from the Manufacturer there will be a number of predefined Templates and at least one SP (the Admin SP). There may also be additional SPs issued on the TPer during the manufacturing process.

5.4.4.1 Templates and the Admin SP

Template metadata is stored in the Admin SP's Table, Column, and Method tables. When the value of the TemplateID column in the Table, Column, or Method tables is zeroes (Null UID reference) it indicates that the row is a normal Table, Column, or Method table row of the Admin SP. When it is not zeroes it is the UID of a Template, indicating that the Table or Column or Method to be created when an SP is issued using that Template.

Rows with a non-zero TemplateID are readable by Anybody.

The Rows column of the Table table on the Admin SP may be Set by the Issuers authority. The Rows column indicates how many free rows shall be available in the given table after issuance is complete. Since the process of issuing an SP can create rows in various tables, it is simpler to have this indicate "room left for the host application to use" rather than "total space to allocate."

5.4.4.2 Admin SP Sessions

An open Read-Write session to the Admin SP shall not be able to be combined with sessions of any type open to any other SPs on the TPer (including sessions that are already open when the attempt to open a Read-Write session to the Admin SP is made).

5.4.4.2.1 Issuance Sessions

Issuance requires a session to the Admin SP that incorporates HostSigning, HostExchange, SPExchange, and optionally SPSigning, all based on Manufacturer controlled Certificates. The Admin SP shall require that the HostSigningAuthority and the HostExchangeAuthority (which may be different) are present in the `ORG` section of the `Authority` Table. Certificate chain down is possible, to the `Chain Limit`.

The critical method in issuance is **`IssueSP`**. The ACL on this method contains exactly one ACE that cannot be changed, and it requires a Boolean combination of Authorities: (HostSigning AND HostExchange AND SPExchange). The SPSigningAuthority is optional but recommended. Issuance will not be deemed completed until the method has completed successfully the session has successfully closed.

During issuance, the host is responsible for providing logging, and fetching information it requires to confirm the issuance.

5.4.4.3 Authorities

The authorities that shall be required by the Admin Template are enumerated in Table 124.

Table 124 Default Admin Template Authorities

Name	UID	Common Name	IsClass	Class
Issuers	00 00 00 09 00 00 02 01	SPControl	True	
Editors	00 00 00 09 00 00 02 02	SPControl	True	
Deleters	00 00 00 09 00 00 02 03	SPControl	True	
Servers	00 00 00 09 00 00 02 04	SPControl	True	
Reserve0	00 00 00 09 00 00 02 05	SPControl	True	
Reserve1	00 00 00 09 00 00 02 06	SPControl	True	
Reserve2	00 00 00 09 00 00 02 07	SPControl	True	
Reserve3	00 00 00 09 00 00 02 08	SPControl	True	

The TPerExch allows a secure session to be established immediately with the Admin SP. Note the corresponding credentials contain certificate chains that establish the validity of TPerSign and TPerExch signed by the manufacturer.

In addition to the authorities defined in Table 124, if a TPer supports Issuance, then it shall be required that the Admin SP have up to an additional 2^16 entries in this table, in blocks of 16, starting immediately after the default Base and Admin Template authorities. These are called ORG authority blocks.

ORG0 is the ORG (anization) of the manufacturer or SP licensing authority. Other ORGs may include other SP licensing authorities. Note the classes include SP Issuance authorities (Issuers), SP ORG Editors that can edit values within an ORG block, SP Deleters that are restricted to deleting ORG authorities within a 16 block, and SP Servers that are used to set up confidential messaging between

Issuance participants. Members of the Servers class are Sign and Exchange authorizations in order to permit secure messaging.

For example values for the Admin Template's required authorities see Table 133.

5.4.4.4 Default Logging Settings

The default logging settings associated with the Admin Template methods are:

- The default logging for Admin SP method, `IssueSP`, is LogAlways.
- All other methods that apply to the Admin SP will be as described in the Base Template reference section (See Section 5.3.4.4).

5.4.5 Life Cycle

5.4.5.1 Admin Template-Specific Life Cycle State Descriptions/Exceptions

The Admin SP has the following characteristics based on the current life cycle state of that SP:

- **Issued –** The SP will have the default Admin Template-related access control settings as described in the following sections.
- **Disabled –** Access control shall prevent the Admin SP from entering the Disabled state.
- **Frozen –** Access control shall prevent the Admin SP from entering the Frozen state.
- **Issued-Disabled-Frozen** – Access control shall prevent the Admin SP from entering the Issued-Disabled-Frozen state.

5.4.5.2 Initial Access Control Settings

The following sections enumerate the initial required access control settings for the table/method combinations provided to an SP by the Admin Template. Note that in the Admin SP, there is no personalization possible.

In the descriptive tables in this section, "None" indicates that the relevant ACL column of the `Method` table has a Null UID reference. This indicates that access control to perform that action cannot be satisfied.

Some methods do not appear in the descriptive tables in this section for some Template tables or objects. This indicates that the method shall not be able to be invoked on that table or object, and there shall be no row in the `Method` table representing that access control association.

5.4.5.2.1 ACEs

In addition to the ACEs defined in the Base Template (see Table 74), which are defaults for all SPs, the following table defines the ACEs added for use in the life cycle of the Admin Template.

Table 125 Admin Template Added ACEs

UID	Name	BoolExpr	RowStart	RowEnd	ColStart	ColEnd
00 00 00 08 00 00 02 01	SID	SID			Frozen	Frozen
00 00 00 08 00 00 02 02	Issuers	ORG*-1 and ORG*-2 and TPerExch			MinSize	MinSize
00 00 00 08 00 00 02 03	Editors	Editors	*	*		
00 00 00 08 00 00 02 04	Deleters	Deleters	*	*		
00 00 00 08 00 00 02 05	Servers	Servers	*	*		

UID	Name	BoolExpr	RowStart	RowEnd	ColStart	ColEnd
00 00 00 08 00 00 02 06	Issuers_SID**	ORG*-1 and ORG*-2 and TPerExch and SID				

*Identifies ACEs that apply to modification of the Authority table on the Admin SP. Their uses are reflected in 5.4.4.3.

ORG*-1 represents an authority that is a signing authority that is a member of the Issuers class and ORG*-2 represents an authority that is an exchange authority that is a member of the Issuers class. ORG*-1 and ORG*-2 shall both belong to the same "ORG". See 5.4.4.3.

** Some SSCs/ORGs may require that the SID be required for Issuance.

5.4.5.2.2 Authority Table Access Control Settings

The `Authority` table on the Admin SP has the following default access control settings. These settings are used in place of the settings normally enabled by the Base Template.

Table 126 Authority Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL
Get	Anybody	None	None	Anybody
Next	Anybody	None	None	Anybody

In addition to the default access control settings defined in Table 126, certain authority objects have the following access capabilities/restrictions:

- The Editors ACE is in the ACL for the `Set` method on all ORG objects in the `Authority` table.
- The Deleters ACE is in the ACL for the `Set` method on the Enabled column on all ORG objects in the `Authority` table.

5.4.5.2.3 Table Table Access Control Settings

On the Admin SP, the `Table` table has these access control settings:

- The Issuers ACE is in the ACL for the `Set` method, which enables the authorities listed in the Issuers ACE to modify the default amount of rows (defined in the `Table` table's `MinSize` column) that will be created for each table when a new SP is issued. This applies only to Table descriptor objects that are not identified as "Active".
- Anybody can retrieve information about any row of the `Table` table that has a `TemplateID` other than zeroes.

5.4.5.2.4 IssueSP Method Access Control Settings

For TPers capable of Issuance, access control over the `IssueSP` method is defined in Table 127.

Table 127 IssueSP Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
IssueSP	Issuers	None	None	Anybody	None

For TPers not capable of Issuance, the `IssueSP` method is not available (has no row in the `Method` table).

5.4.5.2.5 TPerInfo Table Default Access Control Settings

Table 128 TPerInfo Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Anybody	None	None	Anybody	None

5.4.5.2.6 CryptoSuite Table Default Access Control Settings

Table 129 CryptoSuite Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Anybody	None	None	Anybody	None
Next	Anybody	None	None	Anybody	None

5.4.5.2.7 Template Table Default Access Control Settings

Table 130 Template Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Anybody	None	None	Anybody	None
Next	Anybody	None	None	Anybody	None

5.4.5.2.8 SP Table Default Access Control Settings

Table 131 SP Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Anybody	None	None	Anybody	None
Next	Anybody	None	None	Anybody	None

5.4.5.2.9 SP Object Default Access Control settings

Each of the SP objects in the `SP` table, except for `SP.UID=0x00 0x00 0x02 0x05 0x00 0x00 0x00 0x01` (the Admin SP), has default access control settings as enumerated in Table 132.

Table 132 SP Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Anybody	None	None	Anybody	None
Delete	SID	None	None	Anybody	None
Set	SID	None	None	Anybody	None

5.4.6 Examples

5.4.6.1 Example Values for Admin Template Authorities

Every Admin SP shall have the authorities identified in Table 124. The specific values of the credentials may differ depending on the capabilities of the TPer on which it exists. Table 133 provides some example customizations of the portions of the Admin SP's `Authority` table that may exist in a particular implementation.

Table 133 Example Authority Settings

Name	Common Name	IsClass	Class	Enabled	Secure	Operation	Credential Table	Cre-dential	Resp Sign	Resp Exch	Log
Issuers	SPControl	True		True							
Editors	SPControl	True		True							
Deleters	SPControl	True		True							
Servers	SPControl	True		True							
Reserve0	SPControl	True		True							
Reserve1	SPControl	True		True							
Reserve2	SPControl	True		True							
Reserve3	SPControl	True		True							
ORG0-1	ORG0	False	10	True	C_AES_128	Sign	RSA2048	3	7	8	LogSuccess
ORG0-2	ORG0	False	13	True	C_AES_128	Exchange	RSA2048	4	7	8	LogSuccess
ORG0-3	ORG0	False	10	True	C_AES_128	Sign	RSA1024	5	6	6	LogSuccess
ORG0-4	ORG0	False	13	True	C_AES_128	Exchange	RSA1024	6	6	6	LogSuccess
ORG0-5	ORG0	False	11	True	C_AES_128	Sign	RSA2048	7	7	8	LogSuccess
ORG0-6	ORG0	False	13	True	C_AES_128	Exchange	RSA2048	8	7	8	LogSuccess
ORG0-7	ORG0	False	12	True	C_AES_128	Sign	RSA2048	9	7	8	LogSuccess
ORG0-8	ORG0	False	13	True	C_AES_128	Exchange	RSA2048	10	7	8	LogSuccess
…	…	…	…	…	…	…	…	…	…	…	…

5.4.6.2 Typical Required CryptoSuite Values

Table 134 Typical Required CryptoSuite Values

CryptoCall	CryptoLen	CryptoOp	Special	Time	Variance
PIN	32	None	False	Not applicable	Not applicable
RSA	1024	Sign	False	< PrK time>	<PrK time>
RSA	1024	Verify	False	< PuK time>	<PuK time>
RSA	1024	KeyGen	False	<time>	<time>
RSA	2048	Sign	True	< PrK time>	<PrK time>
RSA	2048	Verify	True	< PuK time>	<PuK time>
RSA	2048	KeyGen	True	<time>	<time>
EC	192	Sign	False	<PrK time>	<PrK time>
EC	192	Verify	False	< PuK time>	<PuK time>
EC	192	KeyGen	False	<time>	<time>
EC	224	Sign	False	<PrK time>	<PrK time>
EC	224	Verify	False	< PuK time>	<PuK time>
EC	224	KeyGen	False	<time>	<time>
EC	256	Sign	False	<PrK time>	<PrK time>

CryptoCall	CryptoLen	CryptoOp	Special	Time	Variance
EC	256	Verify	False	< PuK time>	<PuK time>
EC	256	KeyGen	False	<time>	<time>
AES	128	Encrypt	False	< Encrypt time, 2^16 bytes>	<Encrypt time 2^16 bytes>
AES	128	Decrypt	False	< Decrypt time, 2^16 bytes>	<Decrypt time 2^16 bytes>
AES	128	KeyGen	False	<time>	<time>
AES	128	Encrypt	True	< Encrypt time, 2^16 bytes>	<Encrypt time 2^16 bytes>
AES	128	Decrypt	True	< Decrypt time, 2^16 bytes>	<Decrypt time 2^16 bytes>
AES	128	KeyGen	True	<time>	<time>
SHA	160	Hash	False	<Hash time, 2^16 bytes>	Not applicable
SHA	256	Hash	False	<Hash time 2^16 bytes>	Not applicable
SHA	384	Hash	False	<Hash time 2^16 bytes>	Not applicable
SHA	512	Hash	False	<Hash time 2^16 bytes>	Not applicable

5.4.6.2.1 Issuance Session Example

As an example, issuance would contain the following commands and results. In this example, UIDs are replaced by names to improve readability.

```
SMUID.StartSession[
       HostSessionID : <a host assigned session number>,
       SPID : "Admin SP",
       Write : True,
       HostChallenge = <random number>,
       HostExchangeAuthority = ORG0-2_UID,
       HostExchangeCert = certificatechain,
       HostSigningAuthority = ORG0-1_UID,
       HostSigningCert = certificate_chain]
=>
```

```
SMUID.SyncSession[
       HostSessionID: <the host assigned session number>,
       SPSessionID: <a TPer assigned session number>,
       SPChallenge = <a TPer generated random number>
       ]
```

```
SMUID.StartTrustedSession[
       HostSessionID: <host session number>,
       SPSessionID: <a TPer assigned session number>,
       HostResponse : <Host uses ORG0-1 Signing Key to Sign Challenge from TPer>,
       HostEncryptsSessionKey : <Host uses TPerExch  Exchange Key to send Session
       on key to TPer>
       ]
=>
SMUID.SyncTrustedSession[
       HostSessionID: <host session number>,
       SPSessionID: <TPer session number>,
       SPResponse : <TPer uses TPerSign Signing Key to Sign Challenge from
       Host>,
       SPEncryptSessionKey : <TPer uses ORG0-2 Exchange Key to send Session Key
       to Host>
       ]
```

```
SPID1, Size1 = SPUID.IssueSP ['AliceSP, 128, ['Clock' ], <128 bit AES key 1>, true]

SPID2, Size2 = SPUID.IssueSP ['BobSP", 1024, ['Clock', 'Log' ], <128 bit AES key 2>,
true]

SPID3, Size3 = SPUID.IssueSP ['EveSP", 2048, ['Clock', 'Crypto', 'Log' ], <128 bit
AES key 3>, true]

       .
       .
       .

End of Session
```

Upon successfully issuing this set of commands and closing the session, the new SPs are assumed to be ready and operational. The host, or another host on the network, may at this point open sessions to personalize the new SPs: AliceSP, BobSP, and EveSP

5.5 Clock Template

5.5.1 Overview

The Clock Template enables an SP to manage information about time. A TPer may support any number of SPs that incorporate the Clock Template.

5.5.2 Terminology

Table 135 Clock Template Terminology

Term	Definition
ExactTime	ExactTime is a time value represented by the clock_time type. ExactTime is a return value in the GetClock method and a parameter of the SetClockHigh and SetClockLow methods.
HighTime	HighTime represents the actual current High Trust time value, which is the value of the HighSetTime column plus the time elapsed on the IncrementalClock since the HighSetTime value was set. This is the value returned as ExactTime when GetClock is invoked and High Trust time is returned.
High Trust	A time value retrieved from a remote but strongly protected source of time
IncrementalClock	Each Clock Template-enabled SP will have an incremental clock that is accessible from the TPer and is used to measure time intervals.
LagTime	The time period recorded by the Host Application between when it read time from its time source and when it received the OK result from the SP upon successful receipt and processing of the ExactTime parameter of the SetClockHigh or SetClockLow method.
LowTime	LowTime represents the actual current Low Trust time value, which is the value of the LowSetTime column plus the time elapsed on the IncrementalClock since the LowSetTime value was set. This is the value returned as ExactTime when GetClock is invoked and Low Trust time is returned.
Low Trust	An immediate but not strongly protected source of time, such as the local PC clock
MonotonicTime	A 64-bit persistent counter needed for clock requests that require a counter. MonotonicTime operations increment the counter independent of transactions and of the Read/Write state of the session.
MonotonicIncrement	This is a counter kept in main memory that is used to reduce the number of writes to media that are needed to support the MonotonicTime counter.
Timer Mode	The Clock Template-enabled SP operates in this mode after a power cycle/hardware reset, or if time values have never been set. Retrieving the time in Timer mode returns the value of IncrementalClock and MonotonicTime.

5.5.3 Data Structures

5.5.3.1 ClockTime (Array Table)

The ClockTime table contains exactly one row, with `UID=0x00 0x00 0x04 0x01 0x00 0x00 0x00 0x01`.

Table 136 ClockTime Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	The UID of this row (Read-only)
HaveHigh	boolean	If the value of this column is True, then the values in the High Trust time columns (HighByWhom, HighSetTime, HighInitialTimer, and HighLag) are meaningful. If the value of the TrustMode column is Low or Timer then the value of this column shall be False. (Read-only)
HighByWhom	Authority_ref	Authority that set the High Trust time. This value is valid only if HaveHigh is set to True; otherwise it should be zeroes. This is the uidref to the Authority that is the control authority of the session. (Read-only)
HighSetTime	clock_time	The value of this column is the time set to the value of the ExactTime parameter of the SetClockHigh method when that method is successfully invoked. This value is valid only if HaveHigh is set to True; otherwise it should be zeroes.
HighInitialTimer	clock_time	The value of this column is set to the value of the IncrementalClock when the ExactTime parameter of the SetClockHigh method was received. This value is valid only if HaveHigh is set to True; otherwise it should be zeroes. (Read-only)
HighLag	lag	The value of this column is set by the SetClockHigh method. This value is valid only if HaveHigh is set to True; otherwise it should be zeroes. This represents seconds and fractions of a second. (Read-only)
HaveLow	boolean	If the value of this column is True, then the values in the Low Trust time columns (LowByWhom, LowSetTime, LowInitialTimer, and LowLag) are meaningful. If TrustMode is High or Timer, then this value must be False. (Read-only)
LowByWhom	Authority_ref	Authority that set the Low Trust time. This value is valid only if HaveLow is set to True; otherwise it should be zeroes. This is the uidref to the Authority that is the control authority of the session. (Read-only)
LowSetTime	clock_time	The value of this column is the time set to the value of the ExactTime parameter of the SetClockLow method when that method is successfully invoked. This value is valid only if HaveLow is set to True; otherwise it should be zeroes. (Read-only)
LowInitialTimer	clock_time	The value of this column is set to the value of the IncrementalClock when the ExactTime parameter of the SetClockLow method was received and processed. This value is valid only if HaveLow is set to True; otherwise it should be zeroes. (Read-only)
LowLag	lag	The value of this column is set by the SetClockLow method. This value is valid only if HaveLow is set to True; otherwise it should be zeroes. This represents seconds and fractions of a second. (Read-only)

Column	Type	Description
MonotonicBase	uinteger_8	The monotonic time counter value is periodically saved here. (Read-only)
MonotonicReserve	uinteger_8	The value of this column indicates the frequency that the value of the MonotonicBase column is updated. The value of MonotonicIncrement is added to MonotonicBase whenever MonotonicIncrement == MonotonicReserve. (Read-only)
TrustMode	clock_kind	Controls whether HaveHigh, HaveLow, both, or neither are currently in effect.

5.5.4 Methods

The following section identifies methods that operate on the Clock.

5.5.4.1 GetClock (Table Method)

```
ClockTimeTableUID.GetClock[ ]
=>
[ Kind : clock_kind,
ExactTime : clock_time,
LagTime : lag,
MonotonicTime : uinteger_8 ]
```

This method is used to fetch information about the current time.

If the value of the `HaveLow` column is True and the value of the `HaveHigh` column is False, then the result of the `GetClock` method invocation, in pseudo code, are ["Low", LowTime, LowLag, MonotonicTime].

If the value of the `HaveHigh` column is True, then the result of the `GetClock` method invocation, in pseudo code, are ["High", HighTime, HighLag, MonotonicTime].

If the value of both the `HaveLow` and `HaveHigh` columns is False then the result, in pseudo code, is ["Timer", IncrementalClock, 0, MonotonicTime].

Successful invocation of this method increments the MonotonicTime.

5.5.4.1.1 Fails

- If ClockTime is not the uid of the ClockTime table

5.5.4.2 ResetClock (Table Method)

```
ClockTimeTableUID.ResetClock[ ]
=>
[ Result : boolean ]
```

Successful invocation of this method resets the Clock Template-enabled SP's clock values and puts the SP into Timer mode. This method is invoked automatically when a TPer undergoes a hardware reset/power cycle.

The following properties are set when this method is invoked:

- The `HaveHigh` column of the `ClockTime` table is set to False, and the values of the `HighByWhom`, `HighSetTime`, `HighInitialTimer`, and `HighLag` columns are set to zeroes.
- The `HaveLow` column of the `ClockTime` table is set to False, and the values of the `LowByWhom`, `LowSetTime`, `LowInitialTimer`, and `LowLag` columns are set to zeroes.
- MonotonicIncrement = 0
- `ClockTime.TrustMode` = Timer

- If `ClockTime.MonotonicReserve` == 0x00
  - `ClockTime.MonotonicReserve` = <some small value, e.g. 100>
  - `ClockTime.MonotonicBase` = `ClockTime.MonotonicBase` + `ClockTime.MonotonicReserve`
  - MonotonicBase = `ClockTime.MonotonicBase`
  - MonotonicReserve = `ClockTime.MonotonicReserve`

  Note that this guarantees that the MonotonicTime value always increases (although it may, in a `ResetClock`, skip up to the value of MonotonicReserve).

5.5.4.2.1 Fails

- If ClockTime is not the uid of the ClockTime table

5.5.4.3 SetClockHigh/SetLagHigh (Table Methods)

```
ClockTimeTableUID.SetClockHigh[
       ExactTime : clock_time
]
=>
[ Result : boolean ]

ClockTimeTableUID.SetLagHigh[
       LagTime : lag
]
=>
[ Result : boolean,
LowPreserved: boolean ]
```

This method pair is used to set the time from a High Trust source. The invocation of these methods operate as follows:

1. The host invokes the `SetClockHigh` method on the `ClockTime` table. The **ExactTime** input is received and processed. At this time the value that will eventually be used to set `HighInitialTimer` should be computed by the SP, by reading the IncrementalClock value.
2. The SP returns a Result of "True", indicating that the ExactTime value was received and processed.
3. The host invokes the `SetLagHigh` method on the `ClockTime` table. This method must be the next method invoked by the host after invocation of the `SetClockHigh` method invocation. If it is not, the `SetClockHigh` method's ExactTime values to be stored in the `ClockTime` table will not be saved to the table. When the `SetLagHigh` method is received under this condition, the **LagTime** input is received and processed. If this invocation is accepted, then
   - The value of the `HaveHigh` column is set to True.
   - The value of the `HighByWhom` column is set to the authority for this session.
   - The value of the `HighSetTime` column is set to ExactTime.
   - The value of the `HighInitialTimer` column is set to the previously read IncrementalClock value.
   - The value of the `HighLag` column is set to LagTime.
4. If the new `HighSetTime` and `HighLag` values do not bracket existing `LowSetTime` and `LowLag` values, then the value of the `HaveLow` column is set to False and the values of the `LowByWhom`, `LowSetTime`, `LowInitialTimer`, and `LowLag` columns are set to zeroes. For additional information, see 5.5.5.2.

5. Once all of the above steps have been processed, the SP shall return the method result. If step 3 was completed successfully, the Result is returned as True. If any of the updates in step 3 were not successfully completed, Result is returned as False. If the HaveLow column is not set to False from True and the LowByWhom, LowSetTime, LowInitialTimer and LowLag columns are not set to zeroes due to the described bracketing, then LowPreserved is returned as True. Otherwise, Low is returned as False.

5.5.4.3.1 Fails

- If ClockTimeTableUID is not the uid of the ClockTime table.
- If the SetLagHigh method is not received immediately after the SetClockHigh method.
- If the value of TrustMode is not Low.

5.5.4.4 SetClockLow/SetLagLow (Table Method)

```
ClockTimeTableUID.SetClockLow[
       ExactTime : clock_time,
]
=>
[ Result : boolean ]

ClockTimeTableUID.SetLagLow[
       LagTime : lag
]
=>
[ Result : boolean ]
```

This method pair is used to set the time from a Low Trust source. The invocation of these methods operate as follows:

1. The host invokes the SetClockLow method on the ClockTime table. The **ExactTime** input is received and processed. At this time the value that will eventually be used to set LowInitialTimer should be computed by the SP by reading the IncrementalClock value.
2. The SP returns a Result of "True", indicating that the ExactTime value was received and processed.
3. The host invokes the SetLagLow method on the ClockTime table. This method must be the next method invoked by the host after invocation of the SetClockLow method invocation. If it is not, the SetClockLow method's ExactTime values to be stored in the ClockTime table will not be saved to the table. When the SetLagLow method is received under this condition, the **LagTime** input is received and processed. If this invocation is accepted, then
    - The value of the HaveLow column is set to True.
    - The value of the LowByWhom column is set to the authority for this session.
    - The value of the LowSetTime column is set to ExactTime.
    - The value of the LowInitialTimer column is set to the previously read IncrementalClock value.
    - The value of the LowLag column is set to LagTime
4. If the value of TrustMode is LowAndHigh and HaveHigh is True, then this call shall be accepted only when the existing HighSetTime and HighLag values bracket the new LowSetTime and LowLag values. For additional information, see 5.5.5.2.
5. Once all of the above steps have been processed, the SP shall return the method result. If step 3 was completed successfully and the condition noted in step 4 was met, the updates are made to the ClockTime table and the Result is returned as True. If any of the updates in step 3 were not successfully completed or the condition noted in Step 4 was not met, Result is returned as False.

This invocation is accepted only when the value of the `TrustMode` column is not “High”.

5.5.4.4.1 Fails

- If `ClockTimeTableUID` is not the uid of the `ClockTime` table.
- If the `SetLagLow` method is not received immediately after the `SetClockLow` method.
- If the values of `HighSetTime` and `HighLag` do not bracket the `SetClockLow` method’s ExactTime combined with the `SetLagLow` method’s LagTime.
- If the value of the TrustMode column is not “High”.

5.5.4.5 IncrementCounter (Table Method)

```
ClockTimeTableUID.IncrementCounter[ ]
=>
[ MonotonicTime : uinteger_8 ]
```

This method increments and then returns the value of the monotonic counter as follows:

if ++MonotonicIncrement == MonotonicReserve

MonotonicBase += MonotonicReserve

MonotonicIncrement = 0

ClockTime.MonotonicBase = MonotonicBase

return MonotonicTime = MonotonicBase + MonotonicIncrement

For two calls to `IncrementCounter`, the later call will always return a value that is greater than that returned by the earlier call.

This method is permitted in a Read-Only session. The incrementing of the counter’s value is not subject to transactional rollback.

5.5.4.5.1 Fails

- If `ClockTimeTableUID` is not the uid of the `ClockTime` table.

5.5.5 Descriptions

The Clock Template enables an SP to keep track of date-time utilizing two time markers:

- A time value called ExactTime that records a time stamp that can be interpreted either in Generalized Time or UTC Time format.
- An error value called LagTime

5.5.5.1 Setting the Time

The SP that incorporates the Clock Template receives the time from a Host Application. It is expected that the Host Application or some process communicating through the Host Application will monitor and record the time lag between the point when the Host Application reads the clock time from the source it is using to get the time, and the point when the Host Application receives confirmation from the SP that the value has been received.

The Host Application then sends to the SP the lag that it has recorded and, on receipt of this value, the SP records both the time and the lag in the `ClockTime` table. In this way, the SP has a value for the time and can also bracket the error.

5.5.5.2 High Trust vs. Low Trust

A distinction is made between time from a High Trust source and time from a Low Trust source. A High Trust source may be a remote but strongly protected source of time. A Low Trust source may be an immediate but not strongly protected source of time, such as the local PC clock.

The High Trust source is expected to be able to provide a more authoritative time, but with a larger lag, so the High Trust source is used to bracket the Low Trust source. In this way, a Low Trust but accurate time may also be detected and used.

When the TrustMode is LowAndHigh and both High Trust and Low Trust values are present, then the Low Trust time is rejected if it isn't confirmed by the High Trust time. Specifically, the following should be true (See 5.5.5.4 for descriptions of LowTime and HighTime):

- LowTime ≥ HighTime.
- LowTime + LowLag ≤ HighTime + HighLag.

If either of these conditions is not true, the Low Trust value is discarded because it is probably wrong. This means that if Low Trust values exist in the `ClockTime` table, a `SetClockHigh` method invocation is received, and either of the above conditions is false, then the Low Trust values are set to 0; or, if High Trust values exist in the `ClockTime` table and a `SetClockLow` method invocation is received, the method invocation fails if either of the above conditions is not true.

The SP that incorporates the Clock Template may accept a Low Trust time with or without an existing High Trust bracket, or just a High Trust time.

5.5.5.3 Monotonic Counter

An SP that incorporates the Clock Template also shall independently maintain a counter that increments every time a clock time is read – this is a 64-bit persistent counter called **MonotonicTime**. The counter is incremented independent of transactions and of the Read/Write state of the session.

This counter is needed for clock requests that require a counter, since it is possible to have the SP time set back in time, and to enable differentiation between multiple requests received at the same clock time.

For each SP that incorporates the Clock Template, there shall also be a counter kept in main memory called **MonotonicIncrement**. This counter is used to reduce the number of writes to media that are needed to support the MonotonicTime.

The value of the virtual variable MonotonicTime that the user sees (via the `IncrementCounter` or `GetClock` methods) will be:

MonotonicTime = MonotonicBase + MonotonicIncrement

The following is always true:

0 ≤ MonotonicIncrement ≤ MonotonicReserve

Note that in this case, MonotonicBase and MonotonicReserve are not necessarily the values stored in the `ClockTime` table. Rather, these values of MonotonicBase and MonotonicReserve are written to media only as needed to guarantee that the `IncrementCounter` method always returns a unique value after a power cycle, etc.

In order to reduce writes to media, the `MonotonicBase` value stored in the `ClockTime` table is only occasionally updated. This is controlled by the value in the `MonotonicReserve` column of the `ClockTime` table.

5.5.5.4 Incremental Clock

Each TPer shall have a quickly accessible incremental clock. This is referred to as **IncrementalClock**. Although this clock will not have the correct absolute time, it will be accurate in measuring time intervals.

To support Host interaction with the Clock Template-enabled SP, two virtual variables – HighTime and LowTime – are used. HighTime and LowTime internally represent actual current time values.

Calculation of the values of HighTime and LowTime uses the original time set (the value of the `HighSetTime` or `LowSetTime` columns), the value of IncrementalClock when those columns were set

(the value of the HighInitialTimer or LowInitialTimer columns), and the current value of IncrementalClock:

HighTime = HighSetTime + (IncrementalClock – HighInitialTimer)

LowTime = LowSetTime + (IncrementalClock – LowInitialTimer)

The HighTime value is changed to new value *v* as follows (as when a SetClockHigh or SetClockLow method invocation is received):

HighSetTime = v

HighInitialTimer = IncrementalClock

The LowTime virtual variable is changed similarly.

This approach avoids the need to update the media as the value of IncrementalClock changes.

Some TPers may also include other special hardware that can be used to implement the Clock Template. These include a real-time clock (with battery backup) and non-volatile memory that can be used to store monotonic counter values.

The existence of a real-time clock on the TPer shall be reported in the response to the Properties method (see section Table 32).

5.5.5.5 Timer Mode

The Clock Template provides an additional time mode, **Timer Mode**, to identify when the time has been un-set after a disk controller reset or if the SP has never had a time set.

After a TPer reset or upon issuance, the SP is in Timer mode. In Timer mode, the time is incremented, but a successful invocation of the GetClock method will return a clock_kind of “Timer”, the values of the IncrementalClock and MonotonicTime, and a LagTime of 0. This indicates that the time value cannot be trusted as an absolute because of the reset.

The ResetClock method is invoked at power up or after a TPer reset, before the SP that incorporates the Clock Template is accessible. This places the TPer into Timer Mode.

If the TPer has a real-time clock, the TPer will use that value at power up or after a TPer reset so long as the real-time clock has power. Otherwise, the TPer reverts to the behavior previously described.

5.5.5.6 Storing Time

The clock_time data type is used to represent time in the ClockTime and other tables. This type may be used to represent either UTC or Generalized time. The clock_time type is a list made up of the following base types (the information in parentheses identifies the data requirement for the value that the TPer shall enforce):

- Year (4 digits) – uinteger_2
- Month (2 digits, 1-12) – uinteger_1
- Day (2 digits, 1-31) – uinteger_1
- Hour (2 digits, 0-23) – uinteger_1
- Minute (2 digits, 0-59) – uinteger_1
- Second (2 digits, 0-59) – uinteger_1
- Fraction (number of milliseconds, 0-999) – uinteger_2

5.5.5.7 Storing LagTime

LagTime is stored in the ClockTime table and represented as a method parameter or return result by a data type that is a list made up of the following base types:

- Seconds – uinteger_2
- Fraction – uinteger_2

5.5.5.8 Default Logging Settings

The default logging settings associated with the Clock Template methods are:

- The default logging for all Clock Template-enabled methods (`ResetClock`, `SetClockHigh`, `SetClockLow`, `IncrementCounter`) is LogAlways.
- All other methods that apply to the `ClockTime` table will be as described in the Base Template reference section (See Section 5.3.4.4).

5.5.6 Life Cycle

5.5.6.1 Clock Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Clock Template has the following characteristics based on the current life cycle state of that SP:

- **Issued –** At Issuance the SP will have the default Clock Template-related access control settings as described in the following sections.
- **Disabled** – If the Clock Template-enabled SP has entered the Disabled state, the SP shall log authentication, session startup, and method invocation attempts, if the Log Template has been issued into the SP. These log entries shall have timestamps of the kind appropriate to that log entry. TPer resets shall cause the SP's `ClockTime` table to revert to Timer mode. Log entries added while the SP is in the Disabled state shall not be retrievable – see other behaviors of SPs in the Disabled state, as described in section 4.4.3.
- **Frozen** – When a Clock Template-enabled SP enters the Frozen state, TPer resets shall still cause the SP's `ClockTime` table to revert to Timer mode. Logging shall not occur while the SP is in the Frozen state. Other behaviors of the SP in the shall be as described in section 4.4.4
- **Issued-Disabled-Frozen** – If the Clock Template-enabled SP has entered the Issued-Disabled-Frozen state, the SP shall log authentication, session startup, and method invocation attempts, if the Log Template has been issued into the SP. These log entries shall have timestamps of the kind appropriate to that log entry. TPer resets shall cause the SP's `ClockTime` table to revert to Timer mode. Log entries added while the SP is in the Issued-Disabled-Frozen state shall not be retrievable. A Clock Template-enabled SP that is in the Issued-Disabled-Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.5.

5.5.6.2 Initial Access Control Settings

The following sections enumerate the initial required access control settings for the table/method combinations provided to an SP by the Clock Template. These access controls represent the pre-personalization settings of the Clock Template-related table/method combinations, i.e. those that exist when the SP first enters the Issued state.

In the descriptive tables in this section, "None" indicates that the relevant ACL column of the `Method` table has a Null UID reference (zeroes). This indicates that access control to perform that action cannot be satisfied.

Some methods do not appear in the descriptive tables in this section for some Template tables or objects. This indicates that the method shall not be able to be invoked on that table or object, and there shall be no row in the `Method` table representing that access control association.

5.5.6.2.1 ACEs

There are no ACEs added to support the Clock Template.

5.5.6.2.2 ClockTime Table Default Access Control Settings

Table 137 ClockTime Table Default Access Control

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Admins	Admins	Admins	Admins	Admins
GetClock	Admins	Admins	Admins	Admins	Admins
IncrementCounter	Admins	Admins	Admins	Admins	Admins
ResetClock	Admins	Admins	Admins	Admins	Admins
SetClockHigh	Admins	Admins	Admins	Admins	Admins
SetLagHigh	Admins	Admins	Admins	Admins	Admins
SetClockLow	Admins	Admins	Admins	Admins	Admins
SetLagLow	Admins	Admins	Admins	Admins	Admins

5.5.7 Examples

5.5.7.1 Example ClockTime Tables

Table 138 Example ClockTime Table 1 – High Trust Time

UID	Have High	High ByWhom	HighSetTime	High InitialTimer	HighLag	Have Low	Low ByWhom
00 00 04 01 00 00 00 01	True	9	{Year=2006, Month=8, Day=2, Hour=11, Minute=58, Second=37}	{Year=2006, Month=1, Day=2, Hour=11, Minute=58, Second=37, Fraction=26}	{Seconds=135, Fraction=900}	False	0

LowSetTime	Low InitialTimer	LowLag	Monotonic Base	Monotonic Reserve	Trust Mode
0	0	0	1	3	High

Table 139 Example ClockTime Table 2 – Low Trust Time

UID	Have High	High ByWhom	HighSetTime	High InitialTimer	HighLag	Have Low	Low ByWhom
00 00 04 01 00 00 00 01	False	0	0	0	0	True	9

LowSetTime	Low InitialTimer	LowLag	Monotonic Base	Monotonic Reserve	Trust Mode

LowSetTime	Low InitialTimer	LowLag	Monotonic Base	Monotonic Reserve	Trust Mode
{Year=2006, Month=8, Day=2, Hour=11, Minute=59, Second=18}	{Year=2006, Month=1, Day=2, Hour=11, Minute=59, Second=18, Fraction=91}	{Seconds=16, Fraction=80}	1	3	Low

Table 140 Example ClockTime Table 3 – High and Low Trust Time

UID	Have High	High ByWhom	HighSetTime	High InitialTimer	HighLag	Have Low	Low ByWhom
00 00 04 01 00 00 00 01	True	9	{Year=2006, Month=8, Day=2, Hour=11, Minute=58, Second=37}	{Year=2006, Month=1, Day=2, Hour=11, Minute=58, Second=37, Fraction=26}	{Seconds=135, Fraction=900}	True	9

LowSetTime	Low InitialTimer	LowLag	Monotonic Base	Monotonic Reserve	Trust Mode
{Year=2006, Month=8, Day=2, Hour=11, Minute=59, Second=18}	{Year=2006, Month=1, Day=2, Hour=11, Minute=59, Second=18, Fraction=91}	{Seconds=16, Fraction=80}	4	3	LowAndHigh

Table 141 Example ClockTime Table 3 – Timer

UID	Have High	High ByWhom	HighSetTime	High InitialTimer	HighLag	Have Low	Low ByWhom
00 00 04 01 00 00 00 01	False	0	0	0	0	False	0

LowSetTime	Low InitialTimer	LowLag	Monotonic Base	Monotonic Reserve	Trust Mode
0	0	0	4	3	Timer

5.6 Crypto Template

5.6.1 Overview

The Crypto Template provides a set of cryptographic methods that operate on public and symmetric key store tables, collectively called Credential tables, provided by the Base and other Templates. The Crypto Template also provides a set of tables that supports these methods.

The set of cryptographic methods that the Crypto Template provides support functionality that includes Encryption, Decryption, Signing, Verifying, Hashing, HMAC, and XOR. Other Templates may provide Credential tables to an SP. Credential tables in an SP that does not incorporate the Crypto Template may be key stores or contain credentials for use in media encryption, secure messaging, and authentication. Incorporating the Crypto Template into an SP enables the host to perform encryption, decryption, signing, and verification on the TPer, using keys and data stored on the TPer.

5.6.2 Terminology

Table 142 Crypto Template Terminology

Term	Definition
"stream"	The term "stream", used in quotation marks in this section, is not related to session or messaging streams. Rather, this term is used to identify a single operational context related to a particular cryptographic operation. A "stream" is created using an initialization method, is operated on by one or more calculation methods of the type appropriate to the initialized "stream", and is closed by a finalization method. A particular context shall deal only with the operation associated with it (encrypt, decrypt, HMAC, or hash).

5.6.3 Data Structures

The Crypto Template provides tables similar to the Credential tables described by the Base and other Templates. However, unlike those Credential tables, which represent key stores or authentication associations, the Crypto Template's tables are Credential support tables optimized for incremental on-TPer operations.

5.6.3.1 Cryptographic Support Group - H_SHA_1 (Object Table)

This section describes the support table for use with SHA-1 hashing operations.

Table 143 H_SHA_1 Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier for this row. (Read-only)
Name	Yes	name	Name of this object (Read-only for pre-personalization objects)
CommonName	Yes	name	A convenient name that may be shared by several objects. (Read-only for pre-personalization objects)
Proof		bytes_20_def_00	This is the proof to be checked when the Verify method is invoked on this credential object; or this is the proof to be created when the Sign method is invoked on this Credential object. The default value of this column is 00s.

Column	IsIndex	Type	Description
Accumulator		bytes_20_def_00	This is the accumulator where a new hash is incrementally created upon invocation of the Hash or HMAC methods on this credential, or where an initial condition is set. The default value of this column is zeroes.
Signer		cred_object_uidref	Reference to the signing/verification credential object. This is the signing credential whose public key or symmetric key will decrypt the proof to reveal the proof's underlying hash when the Verify method is invoked; or whose private or symmetric key will encrypt the proof when the Sign method is invoked on this credential; or whose HMAC key will be used when the HMAC methods are invoked on this credential. The default value of this column is zeroes (Null UID reference).

Object of this table are used with the `HashInit`, `HashCalc`, `HashFinalize`, `HMACInit`, `HMACCalc`, `HMACFinalize`, `Sign`, and `Verify` methods.

5.6.3.2 Cryptographic Support Group - H_SHA_256 (Object Table)

This section describes the support table for use with SHA-256 hashing operations.

Table 144 H_SHA_256 Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier for this row. (Read-only)
Name	Yes	name	Name of this object (Read-only for pre-personalization objects)
CommonName	Yes	name	A convenient name that may be shared by several objects. (Read-only for pre-personalization objects)
Proof		bytes_32_def_00	This is the proof to be checked when the Verify method is invoked on this credential object; or this is the proof to be created when the Sign method is invoked on this Credential object. The default value of this column is 00s.
Accumulator		bytes_32_def_00	This is the accumulator where a new hash is incrementally created upon invocation of the Hash or HMAC methods on this credential, or where an initial condition is set. The default value of this column is zeroes.
Signer		cred_object_uidref	Reference to the signing/verification credential object. This is the signing credential whose public key or symmetric key will decrypt the proof to reveal the proof's underlying hash when the Verify method is invoked; or whose private or symmetric key will encrypt the proof when the Sign method is invoked on this credential; or whose HMAC key will be used when the HMAC methods are invoked on this credential. The default value of this column is zeroes (Null UID reference).

Object of this table are used with the `HashInit`, `HashCalc`, `HashFinalize`, `HMACInit`, `HMACCalc`, `HMACFinalize`, `Sign`, and `Verify` methods.

5.6.3.3 Cryptographic Support Group - H_SHA_384 (Object Table)

This section describes the support table for use with SHA-384 hashing operations.

Table 145 H_SHA_384 Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier for this row. (Read-only)
Name	Yes	name	Name of this object (Read-only for pre-personalization objects)
CommonName	Yes	name	A convenient name that may be shared by several objects. (Read-only for pre-personalization objects)
Proof		bytes_48_def_00	This is the proof to be checked when the Verify method is invoked on this credential object; or this is the proof to be created when the Sign method is invoked on this Credential object. The default value of this column is 00s.
Accumulator		bytes_48_def_00	This is the accumulator where a new hash is incrementally created upon invocation of the Hash or HMAC methods on this credential, or where an initial condition is set. The default value of this column is zeroes.
Signer		cred_object_uidref	Reference to the signing/verification credential object. This is the signing credential whose public key or symmetric key will decrypt the proof to reveal the proof's underlying hash when the Verify method is invoked; or whose private or symmetric key will encrypt the proof when the Sign method is invoked on this credential; or whose HMAC key will be used when the HMAC methods are invoked on this credential. The default value of this column is zeroes (Null UID reference).

Object of this table are used with the `HashInit`, `HashCalc`, `HashFinalize`, `HMACInit`, `HMACCalc`, `HMACFinalize`, `Sign`, and `Verify` methods.

5.6.3.4 Cryptographic Support Group - H_SHA_512 (Object Table)

This section describes the support table for use with SHA-512 hashing operations.

Table 146 H_SHA_512 Table Description

Column	IsIndex	Type	Description
UID		uid	Unique identifier for this row. (Read-only)
Name	Yes	name	Name of this object (Read-only for pre-personalization objects)
CommonName	Yes	name	A convenient name that may be shared by several objects. (Read-only for pre-personalization objects)
Proof		bytes_64_def_00	This is the proof to be checked when the Verify method is invoked on this credential object; or this is the proof to be created when the Sign method is invoked on this Credential object. The default value of this column is 00s.

Column	IsIndex	Type	Description
Accumulator		bytes_64_def_00	This is the accumulator where a new hash is incrementally created upon invocation of the Hash or HMAC methods on this credential, or where an initial condition is set. The default value of this column is zeroes.
Signer		cred_object_uidref	Reference to the signing/verification credential object. This is the signing credential whose public key or symmetric key will decrypt the proof to reveal the proof's underlying hash when the Verify method is invoked; or whose private or symmetric key will encrypt the proof when the Sign method is invoked on this credential; or whose HMAC key will be used when the HMAC methods are invoked on this credential. The default value of this column is zeroes (Null UID reference).

Object of this table are used with the `HashInit`, `HashCalc`, `HashFinalize`, `HMACInit`, `HMACCalc`, `HMACFinalize`, `Sign`, and `Verify` methods.

5.6.4 Methods

5.6.4.1 Key Related Method Group - Random (SP Method)

This section describes the method used to generate random numbers.

```
SPUID.Random[
       Count : uinteger_256,
       BufferOut = cell_block ]
=>
[ Result : max_bytes_256 ]
```

This method returns a sequence of random bytes of a specified size, as defined in the **Count** method parameter. The quality of random numbers generated is under the purview of the conformance profile.

If the **BufferOut** parameter is specified, **Result** will be empty.

5.6.4.2 Crypto Related Method Group – Stir (SP Method)

```
SPUID.Stir[
       Value : stir_input
       ]
=>
[ Result : boolean ]
```

The purpose of this method is to add additional information for use by the `Random` method for subsequent invocations of that method.

The `Stir` method is invoked using the **Value** parameter, which can hold either an integer or a Boolean value.

Invocation of the `Stir` method with the integer Value parameter allows the host to pass an integer of its choice as the information to be added for use by the `Random` method.

Invocation of the `Stir` method with the Boolean Value parameter indicates that the TPer should generate the information to be used by the `Random` method.

Invocation of the `Stir` method with a Value parameter of False is an error.

5.6.4.3 Decryption Method Group – DecryptInit (Object Method)

```
CredentialObjectUID.DecryptInit[
       IV = iv  ]
=>
[ Result : boolean]
```

This method is used to initiate a decryption "stream" using the credential object that invoked the method. Only one decryption "stream" shall be able to be open at any one time for any individual credential object.

If the **IV** parameter is included, the parameterized IV is used in place of that which may be stored in the credential object itself.

5.6.4.3.1 Fails

- If the object does not exist
- If the object does not contain a valid credential
- If the object is not a symmetric credential
- If the object currently already has a decryption "stream" open

5.6.4.4 Decryption Method Group - Decrypt (Object Method)

```
CredentialObjectUID.Decrypt[
       DataInput : decrypt_input,
       BufferOut = cell_block  ]
=>
[ Result : decrypt_result ]
```

The `Decrypt` method causes the TPer to perform decryption on the data supplied using the credential object that invoked the method.

This method shall require that the `DecryptInit` method has been invoked previously during the session to start a decryption "stream" for the invoking credential object, and that the `DecryptFinalize` method has not yet been invoked to close that "stream". Invoking the `Decrypt` method when there is no open decryption "stream" (i.e. before `DecryptInit` or after `DecryptFinalize`) shall result in an error.

The value of the **DataInput** argument can be either a max bytes value, wherein bytes to be decrypted are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be decrypted. The value passed upon invocation of the `Decrypt` method is decrypted using the key in the specified credential.

If the **Result** cellblock is specified, the DataInput byte length shall be equal in size to or smaller than the cellblock specified for the Result.

The required length of the DataInput bytes is dependant upon the mode of operation selected for the credential. Should padding be required, the host shall perform it.

If the host invokes the `Decrypt` method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the `Decrypt` method, the host must also fulfill the access control required to invoke the `Get` method on the entirety of that cellblock.

If the host invokes the `Decrypt` method using the **BufferOut** cellblock as the target for the result bytes, then in addition to fulfilling the access control on the `Decrypt` method, the host must also fulfill the access control required to invoke the `Set` method on the entirety of that cellblock.

If the BufferOut parameter is specified, the method Result shall be empty.

5.6.4.4.1 Fails

- If the object does not exist
- If the object does not contain a valid credential
- If the object is not a symmetric credential

- If the DataInput cellblock reference is not a to valid cellblock
- If the DataInput is a cellblock reference and Get access control on that cellblock has not been fulfilled
- If the BufferOut is not a valid cellblock
- If BufferOut has been specified and Set access control on that cellblock has not been fulfilled
- If the DataInput byte size is not the same size as or smaller than the Result cell size (if specified).
- If Decrypt has been invoked when no decryption "stream" is open

5.6.4.5 Decryption Method Group – DecryptFinalize (Object Method)

```
CredentialObjectUID.DecryptFinalize[
  ]
=>
[ Result : decrypt_result ]
```

Invocation of this method closes the decryption "stream" associated with this object.

5.6.4.5.1 Fails

- If the object does not exist
- If the object does not contain a valid credential
- If the object is not a symmetric credential
- If there is no decryption "stream" open for this credential object

5.6.4.6 Encryption Method Group – EncryptInit (Object Method)

```
CredentialObjectUID.EncryptInit[
       IV = iv ]
=>
[ Result : boolean]
```

This method is used to initiate an encryption "stream" using the credential object that invoked the method. Only one encryption "stream" shall be able to be open at any one time for any individual credential object.

If the **IV** parameter is included, the parameterized IV is used in place of that which may be stored in the credential object itself.

5.6.4.6.1 Fails

- If the object does not exist
- If the object does not contain a valid credential
- If the object is not a symmetric credential
- If the object currently already has an encryption "stream" open

5.6.4.7 Encrytion Method Group - Encrypt (Object Method)

```
CredentialObjectUID.Encrypt[
       DataInput : encrypt_input,
       BufferOut = cell_block  ]
=>
[ Result : encrypt_result ]
```

The `Encrypt` method causes the TPer to perform encryption on the data supplied using the credential object that invoked the method.

This method shall require that the `EncryptInit` method has been invoked previously during the session to start an encryption "stream" for the invoking credential object, and that the `EncryptFinalize` method has not yet been invoked to close that "stream". Invoking the `Encrypt` method when there is no open encryption "stream" (i.e. before `EncryptInit` or after `EncryptFinalize`) shall result in an error.

The value of the **DataInput** argument can be either a max bytes value, wherein bytes to be encrypted are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be encrypted. The value passed upon invocation of the `Encrypt` method is encrypted using the key in the specified credential.

If the **Result** cellblock is specified, the DataInput byte length shall be equal in size to or smaller than the cellblock specified for the Result.

The required length of the DataInput bytes is dependant upon the mode of operation selected for the credential. Should padding be required, the host shall perform it.

If the host invokes the `Encrypt` method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the `Encrypt` method, the host must also fulfill the access control required to invoke the `Get` method on the entirety of that cellblock.

If the host invokes the `Encrypt` method using the **BufferOut** cellblock as the target for the result bytes, then in addition to fulfilling the access control on the `Encrypt` method, the host must also fulfill the access control required to invoke the `Set` method on the entirety of that cellblock.

If the BufferOut parameter is specified, the method Result shall be empty.

5.6.4.7.1 Fails

- If the object does not exist
- If the object does not contain a valid credential
- If the object is not a symmetric credential
- If the DataInput cellblock reference is not a to valid cellblock
- If the DataInput is a cellblock reference and Get access control on that cellblock has not been fulfilled
- If the BufferOut is not a valid cellblock
- If BufferOut has been specified and Set access control on that cellblock has not been fulfilled
- If the DataInput byte size is not the same size as or smaller than the Result cell size (if specified).
- If Encrypt has been invoked when no encryption “stream” is open

5.6.4.8 Encryption Method Group – EncryptFinalize (Object Method)

```
CredentialObjectUID.EncryptFinalize[
  ]
=>
[ Result : boolean]
```

Invocation of this method closes the encryption “stream” associated with this object.

5.6.4.8.1 Fails

- If the object does not exist
- If the object does not contain a valid credential
- If the object is not a symmetric credential
- If there is no encryption “stream” open for this credential object

5.6.4.9 Sign (Object Method)

```
CredentialObjectUID.Sign
HashObjectUID.Sign[
       DataInput : sign_input,
       BufferOut = cell_block ]
=>
[ Result : sign_result ]
```

This method is used to sign a data input using the private part of a public-private key pair.

The value of the **DataInput** argument can be either a max bytes value, wherein bytes to be signed are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be signed.

For the `Sign` method invoked on an asymmetric credential object, the DataInput value is signed using the private part of the key pair of the specified public key credential.

For the `Sign` method invoked on a hash object, the data in the hash object's `Accumulator` column is signed using the private part of the key pair of the public key credential referenced in the hash object's `Signer` column. It is an error for the `Sign` method to be invoked on a hash object and have the DataInput parameter specified.

If the DataInput parameter specifies a cellblock address as the input to the method, in addition to fulfilling the access control requirements to invoke the `Sign` method, the host shall also fulfill the access control requirements necessary to invoke the `Get` method on the entirety of the specified cellblock.

If the **BufferOut** parameter is specified as the target of the method result, in addition to fulfilling the access control requirements to invoke the `Sign` method, the host shall also fulfill the access control requirements necessary to invoke the `Set` method on the entirety of the specified cellblock.

If the BufferOut parameter is specified, the method **Result** shall be empty.

5.6.4.9.1 Fails

- If the invoking object does not exist
- If the `Sign` method is invoked on a hash object and that object does not reference a valid public key credential (RSA, EC)
- If the invoking credential, or the credential referenced from the hash object, does not contain a valid private key
- If the DataInput cellblock reference is not a to valid cellblock
- If the host application has not fulfilled the access control requirements necessary to invoke the `Get` method on the DataInput cellblock
- If the BufferOut is not a valid cellblock
- If the host application has not fulfilled the access control requirements necessary to invoke the `Set` method on the BufferOut cellblock

5.6.4.10 Verify (Object Method)

```
CredentialObjectUID.Verify
HashObjectUID.Verify[
       DataInput : verify_input,
       Proof = verify_proof ]
=>
[ Result : boolean ]
```

This method may be invoked on a hash object or a public key credential. It is used to verify a signed hash against a proof.

The value of the **DataInput** argument can be either a max bytes value, wherein bytes to be verified are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be verified.

The value of the **Proof** parameter identifies the data that the DataInput is to be compared against. The value of the Proof parameter can be either a max bytes value, wherein the proof data bytes are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the proof data.

5.6.4.10.1 Fails

- If the invoking object does not exist.
- If the invoking credential object is not a valid public key credential (RSA, EC).

- If the invoking hash object does not reference a public key credential.
- If the host application has not fulfilled the access control requirements necessary to invoke the `Get` method on the DataInput or Proof cellblock.
- If the DataInput or Proof are not valid cellblocks.

5.6.4.11 Hash Method Group – HashInit (Object Method)

```
HashObjectUID.HashInit[
       BufferOut = cell_block ]
=>
[ Result : boolean ]
```

This method is used to initiate a hash "stream" using the hash object that invoked the method. Only one hash "stream" shall be able to be open at any one time for any individual hash object.

Invocation of this method is required before the `HashCalc` method can be successfully invoked. In preparation for beginning the hash "stream", upon successful invocation of the `Hashinit` method, the invoking hash object's `Accumulator` column is set to zero.

If the **BufferOut** cellblock is specified, that cellblock shall be larger than or equal to the size of the hash calculation result.

5.6.4.11.1 Fails

- If the hash object does not exist
- If BufferOut has been specified and is not a valid cellblock
- If BufferOut has been specified and is not larger than or equal to the size of the hash calculation result
- If BufferOut has been specified and `Set` access control on that cellblock has not been fulfilled
- If the hash object currently already has a hash "stream" open

5.6.4.12 Hash Method Group – HashCalc (Object Method)

```
HashObjectUID.HashCalc[
       DataInput : hash_input
]
=>
[ Result : hash_result ]
```

Invocation of the `HashCalc` method causes the input Data or Buffer to be hashed. The TPer hashes the data on block boundaries as they are reached.

This method shall require that the `HashInit` method has been invoked previously during the session to start a hash "stream" for the invoking hash object, and that the `HashFinalize` method has not yet been invoked to close that "stream". Invoking the `HashCalc` method when there is not an open hash "stream" (i.e. before `HashInit` or after `HashFinalize`) shall result in an error.

The value of the **DataInput** argument can be either a max bytes value, wherein bytes to be hashed are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be hashed.

Results are stored in the BufferOut cellblock of the `HashInit` method, if that parameter is included. Otherwise the results are stored in the invoking hash object's `Accumulator` column. If the BufferOut parameter is specified in the `HashInit` method, the `HashCalc` method Result will be empty.

The method returns the DataInput that has been consumed in the hash as the method result.

5.6.4.12.1 Fails

- If the object does not exist
- If the DataInput cellblock reference is not a to valid cellblock

- If the DataInput parameter references a cellblock and `Get` access control on that cellblock has not been fulfilled
- If `HashCalc` has been invoked when no hash "stream" is open
- If `HashCalc` cannot write to BufferOut cellblock.

5.6.4.13 Hash Method Group – HashFinalize (Object Method)

```
HashObjectUID.HashFinalize[
]
=>
[ Result : hash_result ]
```

Invocation of the `HashFinalize` method causes the TPer to flush the remaining, non-blocked data through the hash and sets the BufferOut cellblock specified in the `HashInit` method. If the BufferOut cellblock was not supplied to the `HashInit` method, the hash result is set to the `Accumulator` column of the invoking hash object.

The method returns the input data that had not yet been consumed by the hash until it had been flushed by this method invocation as the result of the method.

If there is no open hash "stream" for the invoking hash object, the method invocation shall fail.

5.6.4.13.1 Fails

- If the object does not exist
- If the object does not reference a valid symmetric credential object that contains a valid key
- If BufferOut is not a valid cellblock
- If BufferOut is specified and `Set` access control on that cellblock has not been fulfilled
- If `HMACFinalize` has been invoked when no HMAC "stream" is open

5.6.4.14 HMAC Method Group – HMACInit (Object Method)

```
HashObjectUID.HMACInit[
]
=>
[ Result : boolean ]
```

This method is used to initiate an HMAC "stream" using the hash object that invoked the method. Only one HMAC "stream" shall be able to be open at any one time for any individual hash object.

Invocation of this method is required before the `HMACCalc` method can be successfully invoked. In preparation for beginning the HMAC "stream", upon successful invocation of the `HMACInit` method, the invoking hash object's `Accumulator` column is set to zero.

5.6.4.14.1 Fails

- If the hash object does not exist
- If the hash object does not reference a valid symmetric credential object that contains a valid key
- If the hash object currently already has an HMAC "stream" open

5.6.4.15 HMAC Method Group – HMACCalc (Object Method)

```
HashObjectUID.HMACCalc[
       DataInput : hmac_input }
]
=>
[ Result : hmac_result ]
```

Invocation of the `HMACCalc` method causes the **DataInput** parameter value, or the data referenced by the DataInput parameter, to be hashed using the HMAC algorithm with the symmetric key credential referenced from the invoking hash object.

This method shall require that the HMACInit method has been invoked previously during the session to start an HMAC "stream" for the invoking hash object, and that the HMACFinalize method has not yet been invoked to close that "stream". Invoking the HMACCalc method when there is not an open HMAC "stream" (i.e. before HMACInit or after HMACFinalize) shall result in an error.

The value of the DataInput argument can be either a max bytes value, wherein bytes to have the HMAC operation performed are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to have the HMAC operation performed.

The method returns the input Data or input Buffer value that has been consumed in the hash as the method **Result**.

5.6.4.15.1 Fails

- If the object does not exist
- If the object does not reference a valid symmetric credential object that contains a valid key
- If the DataInput cellblock reference is not a to valid cellblock
- If the DataInput is a cellblock reference and Get access control on that cellblock has not been fulfilled
- If HMACCalc has been invoked when no HMAC "stream" is open

5.6.4.16 HMAC Method Group – HMACFinalize (Object Method)

```
HashObjectUID.HMACFinalize[
       BufferOut = cell_block
]
=>
[ Result : hmac_result ]
```

Invocation of the HMACFinalize method causes the TPer to flush the remaining, non-blocked data through the hash, computes the HMAC, and sets the result to the **BufferOut** cellblock. If the BufferOut cellblock has not been supplied, the HMAC result is set to the Accumulator column of the invoking Hash object.

The method returns the input data that had not yet been consumed by the hash until it had been flushed by this method invocation as the result of the method.

If there is no open HMAC "stream" for the invoking Hash object, the method invocation shall fail.

5.6.4.16.1 Fails

- If the object does not exist
- If the object does not reference a valid symmetric credential object that contains a valid key
- If BufferOut is not a valid cellblock
- If BufferOut is specified and Set access control on that cellblock has not been fulfilled
- If the BufferOut cellblock is specified and it is not larger than or equal to the size of the HMAC result
- If HMACFinalize has been invoked when no HMAC "stream" is open

5.6.4.17 XOR (SP Method)

```
SPUID.XOR[
       PatternInput : byte_table_ref,
       DeletePattern : boolean,
       DataInput : xor_input,
       BufferOut = cell_block
]
=>
[ Result : xor_result ]
```

The input data is XORed using the pattern specified in the **PatternInput** parameter. This parameter is a reference to the byte table that stores the pattern.

That **DataInput** argument can be either a max bytes value, wherein bytes to be XORed are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be XORed.

5.6.4.17.1 Fails

- If the PatternInput is not a byte table
- If the DataInput cellblock reference is not a to valid cellblock
- If BufferOut is not a valid cellblock
- If PatternInput is smaller than the input data
- If BufferOut is smaller than the input data
- If associated access control conditions, as described in 5.6.5.4 are not met

5.6.5 Descriptions

5.6.5.1 Cellblocks

Methods of the Crypto Template utilize cellblocks for parameters. Cellblocks are the data type cell_block, and define a set of rows and columns that make up a congruent area of a table. Use of cellblocks as method parameters necessitate special access control condition requirements.

Any cellblock used as input data to a method requires that the host invoking the method satisfy the access control requirement necessary to invoke the `Get` method on the entirety of the parameterized cellblock.

Any cellblock used as an output buffer for a method requires that the host invoking the method satisfy the access control requirement necessary to invoke the `Set` method on the entirety of the parameterized cellblock.

Exceptions or additions to this, such as are required for the `XOR` method's PatternInput parameter, are noted in the method's description.

5.6.5.2 Hashing

Invocation of the `HashInit` method, followed be one or more `HashCalc` method invocations and the `HashFinalize` method on a `H_SHA_*` object, causes the data parameterized in or referenced from the `HashCalc` method invocations to be hashed in the manner described in FIPS 180-2.

A hash "stream" is initiated using the `HashInit` method invoked upon a hash object. Only one hash "stream" shall be open at any one time for any individual hash object. During a session, invoking the `HashInit` method on a hash object after invoking `HashInit` on that object but before invoking the `HashFinalize` method shall cause the second `HashInit` method invocation to fail.

The `HashInit` method shall be invoked prior to invocation of the `HashCalc` method. In preparation for beginning the hash "stream", upon successful invocation of the `HashInit` method, the invoking hash object's `Accumulator` column is set to zero. After invocation of the `HashInit` method, the `Set` method may be used to set an initial condition in the `Accumulator` column. Invoking the `Set` method on the

Accumulator column after the HashInit invocation and after one or more successful HashCalc invocations may cause the final hash result to be an unexpected or inconsistent value.

The HashInit method has a parameter that allows the host to specify a particular cellblock as the target of the hash's final result. This cellblock shall be set to the final hash result upon successful invocation of the HashFinalize method. Access control requirements necessary to permit invocation of the Set method on the entirety of this cellblock shall be fulfilled, or the HashInit method invocation shall fail. If the BufferOut parameter of the HashInit method was used, the cellblock shall be larger than or equal to the size of the expected final hash calculation result, or the HashInit method invocation shall fail.

Successful invocation of the HashCalc method causes the data input to the HashCalc method to be hashed with the value currently stored in the Accumulator column of the invoking hash object. The HashCalc method returns as its result the input data that has been consumed in the hash. Hashing is done at block boundaries appropriate for the hash object type.

The HashCalc method shall accept either bytes passed across the interface as a parameter of the method invocation; or shall identify a cellblock that holds the data to be hashed. If the data is addressed via cellblock, the host shall fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock, or the HashCalc method invocation shall fail.

Upon invocation of the HashFinalize method, the TPer flushes the remaining, non-blocked data through the hash function represented by the invoking hash object. Upon completion of hashing, the final hash result is set to the cellblock specified in the HashInit method. If this parameter was not included in the HashInit method, then the Accumulator column of the invoking hash object is set to the final hash result. The HashFinalize method returns any data that had previously been input but had not yet been hashed and returned in the HashCalc method. The HashFinalize method closes the hash "stream" on the invoking hash object.

If the BufferOut parameter of the HashInit method was used, the referenced cellblock shall be set to the final hash value. Access control requirements necessary to permit invocation of the Set method on the entirety of this cellblock shall be fulfilled, or the HashFinalize method shall fail. If the BufferOut parameter of the HashInit method was used, the cellblock shall be larger than or equal to the size of the expected final hash calculation result, or the HashFinalize method invocation shall fail.

Invoking HashCalc or HashFinalize on a hash object that does not have an open "stream" shall cause that method invocation to fail.

5.6.5.3 HMAC

Invocation of the HMACInit method, followed be one or more HMACCalc method invocations and the HMACFinalize method on a H_SHA_* object, causes the data parameterized in or referenced from the HMACCalc method invocation to have a message authentication code computed on that input data using the H_SHA_* object upon which the method was invoked, the HMAC key referenced from that H_SHA_* object, and the HMAC algorithm described in FIPS 198.

An HMAC "stream" is initiated using the HMACInit method invoked upon a hash object. Only one HMAC "stream" shall be open at any one time for any individual hash object.

During a session, invoking the HMACInit method on a hash object after invoking HMACInit on that object but before invoking the HMACFinalize method shall cause the second HMACInit method invocation to fail. The HMACInit method shall be invoked prior to invocation of the HMACCalc method.

Successful invocation of the HMACCalc method causes the data input to be hashed on block boundaries as they are reached. Intermediate results are stored as internal state and are not accessible from the host.

The HMACCalc method shall accept either bytes passed across the interface as a parameter of the method invocation; or shall identify a cellblock that holds the data to be hashed. If the data is

addressed via cellblock, the host shall fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock, or the HMACCalc method invocation shall fail.

Successful invocation of the HMACCalc method returns the input data that was consumed in the hash method.

Upon invocation of the HMACFinalize method, the TPer flushes the remaining, non-blocked data through the hash function represented by the invoking hash object, computes the HMAC, and sets the result to the BufferOut cellblock if specified. If the BufferOut cellblock is specified, the host shall be required to fulfill access control requirements necessary to successfully invoke the Set method on the entirety of that cellblock.

If the BufferOut cellblock has been specified in the HMACFinalize method invocation, that cellblock shall be larger than or equal to the size of the HMAC calculation result or the HMACFinalize method invocation shall fail.

If the BufferOut cellblock has not been supplied, the HMAC result is set to the Accumulator column of the invoking hash object.

The HMACFinalize method invocation shall return the data that had been supplied as input to the HMACCalc method that had not yet been consumed.

Invoking HMACCalc or HMACFinalize on a hash object that does not have an open “stream” shall cause that method invocation to fail.

5.6.5.4 XOR

Invocation of the XOR method causes the XOR method’s input data to be XORed with the pattern specified in the PatternInput parameter of the method invocation.

The XOR method shall accept either bytes passed across the interface as a parameter of the method invocation; or shall identify a cellblock that holds the data to be hashed. If the data is addressed via cellblock, the host shall fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock, or the XOR method invocation shall fail.

The PatternInput parameter of the method shall be the uid of a byte table that holds the pattern with which the input data shall be XORed. The PatternInput shall be the same size or larger than the input data or the method shall fail. The host shall be required to fulfill access control requirements necessary to invoke the Get method on the entirety of the PatternInput cellblock, or the XOR method invocation shall fail.

The DeletePattern parameter identifies the behavior of the PatternInput table after the XOR operation is complete. If the DeletePattern parameter is True, then at completion of the XOR operation the contents of the byte table referenced as the PatternInput shall be set to all 00’s. If the DeletePattern parameter is False, the method shall not alter the contents of the referenced byte table. The host shall fulfill access control requirements that permit invocation of the Set method on the PatternInput table or the XOR method invocation shall fail.

If the host’s intention is to use the XOR method as a one-time pad, the host should invoke the XOR method with a DeletePattern value of True.

The XOR method returns data in one of two ways. If the BufferOut parameter is specified in the method invocation, then the XOR method result is set to that cellblock. The BufferOut parameter is specified, the cellblock shall be the same size or larger than the XOR result. The host shall be required to fulfill access control requirements necessary to invoke the Set method on the entirety of that cellblock, or the XOR method invocation shall fail. If the BufferOut parameter is specified, the method result shall be empty.

If the BufferOut parameter is not specified, the method result shall be the result of the XOR operation.

5.6.5.5 Signing

Signing of a selected input can be accomplished as described in the following subsections. The exact algorithms used in the signing and verification of digital signatures are defined in FIPS 186-2, and are dependent on the public scheme (RSA, EC, etc.) used.

5.6.5.5.1 Invocation of Sign on a Public Key Credential

Invocation of the `Sign` method can be done by invoking the method on a key pair credential object that has a private key (for example, `C_RSA_1024` or `C_EC_256` objects) and either passing in data or referencing data to be used as the input for signing. The TPer utilizes the private key stored in the credential object referenced in the invocation and performs a private key signing operation on the input data.

For this usage, if a cellblock is referenced to hold the output of the `Sign` method invocation, then no data is returned, and the result of the `Sign` is stored in the referenced cell_block. The host shall fulfill access control requirements necessary to invoke the `Set` method on the entirety of that target cellblock.

If a cellblock is not referenced to hold the `Sign` method output, the data returned is the result of the signing operation performed on the input data with the private key of the referenced credential object.

5.6.5.5.2 Invocation of Sign on a Hash Object

A second way to accomplish signing is to invoke the `Sign` method on a `H_SHA_*` object. If the invocation is done in this manner, then the `H_SHA_*` object upon which the method was invoked shall reference a key pair credential object that has a private key. The signing operation in this case is done using the private key of that referenced credential object.

When invocation of the `Sign` method is done on a `H_SHA_*` object, the signing operation can be performed on either:

1. Data parameterized in or referenced from the `Sign` method invocation. If this is the case, the signed data will either be stored in a cell_block referenced in the invocation or returned as the result of the method invocation. If a cellblock is referenced as the target of the signed data, the host shall be required to fulfill access control requirements necessary to invoke the `Set` method on the entirety of that cellblock, or the `Sign` method invocation shall fail.
2. Or, if no input data or reference is included in the method invocation, the signing operation is performed on the value of the `H_SHA_*` object's `Accumulator` column. If this is the case, the signed data will be stored to the `Proof` column of the `H_SHA_*` object, and can be retrieved with a successful invocation of the `Get` method on that column.

5.6.5.6 Verifying

Verification of a signed hash can be accomplished as described in the following subsections. The exact algorithms used in the signing and verification of digital signatures are defined in FIPS 186-2, and are dependent on the public scheme (RSA, EC, etc.) used.

5.6.5.6.1 Invocation of Verify on a Public Key Credential

Verification may be performed by invoking the `Verify` method on a public key credential.

The `Verify` method is invoked on a public key credential. The proof to be verified against may be supplied in one of two ways.

1. The proof may be parameterized in bytes as the Proof parameter of the `Verify` method invocation.
2. The proof may be stored in a cellblock, which is addressed by the Proof parameter of the `Verify` method.

The value to be verified may be supplied byt the DataInput parameter in one of two ways:

1. The value for verification may be supplied in bytes as the DataInput parameter of the Verify method invocation.
2. The value for verification may be stored in a cellblock, which is addressed by the DataInput parameter of the Verify method.

If either the DataInput or Proof parameters are supplied and address a cellblock, the host shall be required to fulfill the access control requirements necessary to invoke the Get method on the entirety of that cellblock or the Verify method invocation shall fail.

Verification of the input against the proof is performed using the public key of the invoking public key credential.

Invocation of the Verify method shall return True if the verified value matches the proof. Otherwise, the method invocation shall return False.

5.6.5.6.2 Invocation of Verify on a Hash Object

To perform signed hash verification in this way, the Verify method is invoked on a hash object.

The proof to be verified against may be supplied in one of three ways.

1. The proof may be parameterized in bytes as the Proof parameter of the Verify method invocation.
2. The proof may be stored in a cellblock, which is addressed by the Proof parameter of the Verify method.
3. If Proof parameter is not supplied to the Verify method, the proof used shall be the value of the invoking hash object's Proof column.

The value to be verified may also be supplied in one of three ways.

1. The value for verification may be supplied in bytes as the DataInput parameter of the Verify method invocation.
2. The value for verification may be stored in a cellblock, which is addressed by the DataInput parameter of the Verify method.
3. If the DataInput parameter is not supplied to the Verify method, the proof used shall be the value of the invoking hash object's Accumulator column.

If the Proof parameter is supplied and is a cellblock, the host shall be required to fulfill the access control requirements necessary to invoke the Get method on the entirety of that cellblock or the Verify method invocation shall fail.

Verification of the input against the proof is performed using the public key of the public key credential that shall be referenced from the invoking hash object.

Invocation of the Verify method shall return True if the verified value matches the proof. Otherwise, the method invocation shall return False.

5.6.5.7 Encrypting

Invocation of the EncryptInit method, followed by one or more Encrypt method invocations and the EncryptFinalize method, encrypts data that has either been sent to the Crypto template-enabled SP from the host in the method invocation, or that is currently stored in the SP.

Successful invocation of the EncryptInit method is used to initiate an encryption "stream" using the credential object that invoked the method. Only one encryption "stream" shall be open in a session at any one time for a particular credential object. During a session, invoking the EncryptInit method on a credential object after invoking EncryptInit on that object but before invoking the EncryptFinalize method shall cause the second EncryptInit method invocation to fail.

If the optional IV parameter is used in the EncryptInit method, the parameterized IV is used in place of that which may be stored in the ResidualData column of the invoking credential. Otherwise, the value of the ResidualData column of the invoking credential is used as the IV, as required by the value of the credential object's Mode column.

As indicated, the host may generate an initialization vector externally and either pass it as a parameter to the EncryptInit method, or Set the ResidualData column of the symmetric credential object that will be referenced for use with the encryption. Alternatively, the host may invoke the Random method and set its output to the ResidualData column of the symmetric credential object that will be referenced for use with the encryption. For details and guidelines on generation and handling of initialization vectors, reference NIST Special Publication 800-38.

If the EncryptInit method is invoked with an IV and the credential object or credential object's Mode column value do permit use of an IV, then the EncryptInit method shall fail.

The EncryptInit method, upon successful invocation, sets the ResidualData column of the invoking credential to zero.

Successful invocation of the Encrypt method causes the TPer to use the key stored in the invoked credential object to encrypt the input data. Encryption is performed using key stored in the invoking credential object.

The input data may either be parameterized in the Encrypt method invocation, or stored in a table that is referenced as a cellblock from the Encrypt method invocation. If the Encrypt method's DataInput parameter references a cellblock, the host shall fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock or the method invocation shall fail.

If a cellblock is referenced to hold the output of the Encrypt method invocation, then no data is returned, and the result of the Encrypt is stored in the referenced cellblock. If the output cellblock is used, the host shall be required to fulfill the access control requirements necessary to permit invocation of the Set method on the entirety of the referenced cellblock. If the cellblock is not specified, the data returned is the result of the encryption operation performed on the input data.

The length of the data input to the Encrypt method shall be as required by the block size of the particular key type and mode used. Should padding be required, the host shall perform it.

Successful invocation of the Encrypt method causes the invoking symmetric credential object's ResidualData column to have the value specified in Table 48 (for C_AES_128 objects) or Table 50 for C_AES_256 objects).

Upon invocation of the EncryptFinalize method, the encryption "stream" for the invoking credential shall be closed. Invoking Encrypt or EncryptFinalize on a hash object that does not have an open "stream" shall cause that method invocation to fail.

Note that only one write session is open at any given point in time. After closing a write session and opening another write session to the same SP, the host may find that the ResidualData value might have been modified by another write session.

5.6.5.8 Decrypting

Invocation of the DecryptInit method, followed by one or more Decrypt method invocations and the DecryptFinalize method invocation decrypts data that has either been sent to the Crypto template-enabled SP from the host in the method invocation, or that is currently stored in the SP.

Successful invocation of the DecryptInit method is used to initiate a decryption "stream" using the credential object that invoked the method. Only one decryption "stream" shall be open in a session at any one time for a particular credential object. During a session, invoking the DecryptInit method on a credential object after invoking DecryptInit on that object but before invoking the DecryptFinalize method shall cause the second DecryptInit method invocation to fail.

If the optional IV parameter is used in the DecryptInit method, the parameterized IV is used in place of that which may be stored in the ResidualData column of the invoking credential. Otherwise, the value of the ResidualData column of the invoking credential is used as the IV, as required by the value of the credential object's Mode column.

If the DecryptInit method is invoked with an IV and the credential object or credential object's Mode column value do permit use of an IV, then the DecryptInit method shall fail.

The DecryptInit method, upon successful invocation, sets the ResidualData column of the invoking credential to zero.

Successful invocation of the Decrypt method causes the TPer to use the key stored in the invoked credential object to decrypt the input data. Decryption is performed using the key stored in invoking the credential object.

The input data may either be parameterized in the Decrypt method invocation, or stored in a table that is referenced as a cell_block from the Decrypt method invocation. If the Decrypt method's DataInput parameter references a cellblock as the data input, the host shall fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock or the method invocation shall fail.

If a cellblock is referenced to hold the output of the Decrypt method invocation, then no data is returned, and the result of the Decrypt is stored in the referenced cellblock. If the output cellblock is used, the host shall be required to fulfill the access control requirements necessary to permit invocation of the Set method on the entirety of the referenced cellblock. If the cellblock is not specified, the data returned is the result of the decryption operation performed on the input data.

The length of the data input to the Decrypt method shall be as required by the block size of the particular key type and mode used. Should padding be required, the host shall perform it.

Successful invocation of the Decrypt method causes the invoking symmetric credential object's ResidualData column to have the value specified in Table 48 (for C_AES_128 objects) or Table 50 for C_AES_256 objects).

Upon invocation of the DecryptFinalize method, the decryption "stream" for the invoking credential shall be closed. Invoking Decrypt or DecryptFinalize on a hash object that does not have an open "stream" shall cause that method invocation to fail.

Note that only one write session is open at any given point in time. After closing a write session and opening another write session to the same SP, the host may find that the ResidualData value might have been modified by another write session.

5.6.5.9 Default Logging Settings

The default logging settings associated with the Crypto Template methods are:

- The default logging setting for the Delete object method on objects in the H_SHA_* tables, and for invocation of the Verify method, is LogAlways.
- The default setting for all instances of the Crypto Template methods (Sign, HashInit, HashCalc, HashFinalize, HMACInit, HMACCalc, HMACFinalize, XOR, EncryptInit, Encrypt, EncryptFinalize, DecryptInit, Decrypt, and DecryptFinalize) is LogSuccess.
- All other methods that apply to the H_SHA_* tables shall be as described in the Base Template reference section on Default Logging Settings (See 5.3.4.4).

5.6.6 Life Cycle

5.6.6.1 Crypto Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Crypto Template has the following characteristics based on the current life cycle state of that SP:

- **Issued –** At Issuance the SP will have the default Crypto Template-related access control settings as described in the following sections.
- **Disabled –** A Crypto Template-enabled SP that is in the Disabled state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.3.
- **Frozen –** A Crypto Template-enabled SP that is in the Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.4.
- **Issued-Disabled-Frozen** – A Crypto Template-enabled SP that is in the Issued-Disabled-Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.5.

5.6.6.2 Initial Access Control Settings

The following sections enumerate the initial required access control settings for the table/method combinations provided to an SP by the Crypto Template.

In the descriptive tables in this section, “None” indicates that the relevant ACL column of the `Method` table has a Null UID reference. This indicates that access control to perform that action cannot be satisfied.

Some methods do not appear in the descriptive tables in this section for some Template tables or objects. This indicates that the method shall not be able to be invoked on that table or object, and there shall be no row in the `Method` table representing that access control association.

5.6.6.2.1 C_RSA_* Objects Access Control Settings – Crypto Template Methods

All objects created in the C_RSA_* tables have the access control settings associated with Crypto Template methods as defined in Table 147.

Table 147 C_RSA_* Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Sign	Self	Self	Self	Self	Self
Verify	Self	Self	Self	Self	Self

5.6.6.2.2 C_EC_* Objects Access Control Settings – Crypto Template Methods

All objects created in the C_EC_* tables have the access control settings associated with Crypto Template methods as defined in Table 148.

Table 148 C_EC_* Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Sign	Self	Self	Self	Self	Self
Verify	Self	Self	Self	Self	Self

5.6.6.2.3 C_AES_* Objects Access Control Settings – Crypto Template Methods

All objects created in the C_AES_* tables have the access control settings associated with Crypto Template methods as defined in Table 149.

Table 149 C_AES_* Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
EncryptInit	Self	Self	Self	Self	Self
Encrypt	Self	Self	Self	Self	Self
EncryptFinalize	Self	Self	Self	Self	Self
DecryptInit	Self	Self	Self	Self	Self
Decrypt	Self	Self	Self	Self	Self
DecryptFinalize	Self	Self	Self	Self	Self

5.6.6.2.4 H_SHA_* Table/Objects Access Control Settings

Table 150 defines the default access control settings assigned for H_SHA_* tables.

Table 150 H_SHA_* Tables Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	Admins	Admins	Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	Admins	Admins	Admins	Admins	Admins

Table 151 defines the default access control settings for all H_SHA_* objects.

Table 151 H_SHA_* Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
DeleteMethod	Self	Self	Self	Self	Self
Delete	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
HashInit	Self	Self	Self	Self	Self
HashCalc	Self	Self	Self	Self	Self
HashFinalize	Self	Self	Self	Self	Self
HMACInit	Self	Self	Self	Self	Self
HMACCalc	Self	Self	Self	Self	Self
HMACFinalize	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self
Sign	Self	Self	Self	Self	Self
Verify	Self	Self	Self	Self	Self

5.6.7 Examples

5.6.7.1 Example H_SHA_1 Table

Table 152 displays an example of the H_SHA_1 table. This example table is used as a resource for method invocation examples in this section. Values in this table are for illustrative purposes only.

Table 152 Example H_SHA_1 Table

UID	Name	CommonName	Proof	Accumulator	Credential	Signer

HashObject1	Example1	ExGroup	SIGNED HASH	ACCUMULATED HASH	C_EC_256	3
HashObject2	Example2	ExGroup		ACCUMULATED HASH	C_RSA_1024	10
HashObject3	Example3	ExGroup				
HashObject4	Example4	ExGroup			C_AES_128	7

5.6.7.2 Hash Example

The following method invocation series hashes the input data. The hashing result is stored in the `Accumulator` column of the credential upon which the method was invoked. Because no BufferOut was included in the invocation, the result of the method invocation is the return of the input data.

```
HashObject3.HashInit () => [T]
HashObject3.HashCalc (DATA) => [DATA]
HashObject3.HashFinalize () => [DATA]
```

Invoking the `Get` method on the hash credential's `Accumulator` column retrieves the hashed value:

```
HashObject3.Get ([startColumnName = 'Accumulator', endColumnName = 'Accumulator'])
=> [Accumulator = HASHED_DATA]
```

5.6.7.3 HMAC Example

The following method invocation series performs HMAC on the input data. The result is stored in the `Accumulator` column of the credential upon which the method was invoked. Because the BufferOut parameter was included in the invocation, the result of the HMAC operation is stored in the referenced cell_block. Data returned in response to the method invocations is the data that was passed into the methods.

```
HashObject4.HMACInit () => [ ]
HashObject4.HMACCalc (DATA1) => [ DATA1 ]
HashObject4.HMACCalc (DATA2) => [ DATA2 ]
HashObject4.HMACFinalize (BufferOut = [Table='HMACByteTable']) => [ REMAININGDATA ]
```

The `Get` method must be invoked on the `HMACByteTable` in order to retrieve the result of the HMAC operation. .

5.6.7.4 Sign Method Invocation Examples

The following method invocation performs signing by invoking the `Sign` method on a public/private key credential. The input data is signed using the private key in the referenced credential. The signed data is returned as the result of the method invocation.

```
C_RSA_1024Object7.Sign (Data = DATA) => [SIGNED_DATA]
```

The following method invocation performs signing by invoking the Sign method on a `H_SHA_1` object. The `H_SHA_1` object references the credential that performs the signing operation, which in this case is done on the data stored in the `Accumulator` column, and the result of the signing operation is stored in the `Proof` column.

```
HashObject2.Sign ( ) => [ ]
```

The data in the `Proof` column can be retrieved using a `Get` method invocation.

```
HashObject2.Get ([startColumnName = 'Proof, endColumnName = 'Proof']) => [Proof =
SIGNED_DATA]
```

5.6.7.5 Verify Method Invocation Example

This example method invocation performs verification by invoking the `Verify` method on a public/private key credential. The input data is verified against the proof data using the public key in the referenced credential. The method return result is True if the input data matches the proof data. Otherwise the method returns a result of False.

```
C_RSA_2048Object14.Verify (Data = DATA, Proof = PROOF) => [ T ]
```

The following example method invocation displays the operation of the `Verify` method. The method invocation verifies the data in the `Proof` column of the `H_SHA_1` object using the credential referenced from that object.

```
HashObject9.Verify () => [ F ]
```

5.7 Log Template

5.7.1 Overview

The Log Template is designed to maintain a log of the activities on the SP into which it was issued. The purpose of providing this service is to aid in audits, forensic analysis, and general monitoring of the operation of the SP.

An issued SP that incorporates the Log Template should also incorporate the Clock Template to exploit the full capabilities of logging. See Section 5.5 for details on the Clock Template.

5.7.1.1 Terminology

Table 153 Log Template Terminology

Term	Definition
Default log	This is the initial log table created for an SP that incorporates the Log Template. By default, all authority operations, access control associations, transaction events, and session startup events log to this table.

5.7.2 Data Structures

5.7.2.1 Log (Array Table)

Table 154 Log Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	The uid of this log entry. (Read-only)
Session	uinteger_4	Host session number assigned by the TPer. (Read-only)
SigningAuthority	Authority_ref	uid of the Host Signing Authority, if any, that opened the session. (Read-only)
SigningAuthName	name	Name of the Host Signing Authority, if any, that opened the session. (Read-only)
ExchangeAuthority	Authority_ref	uid of the Host Exchange Authority, if any, that opened the session. (Read-only)
ExchangeAuthName	name	Name of the Host Exchange Authority, if any, that opened the session (Read-only)
MonotonicTime	uinteger_8	Monotonic Time value, as defined in section 5.5. Note that if the Clock Template was not issued into this SP then this value will be 0. (Read-only)
ExactTime	clock_time	Exact time (if any) that this log entry was added, as defined in section 5.5. Note that if the Clock Template was not issued into this SP then this value will be zero. (Read-only)
TimeKind	clock_kind	Type of time used (if any), as defined in section 5.5. Note that if the Clock Template was not issued into this SP then this value will be zero. (Read-only)

LogKind	log_kind	This is the user-provided name for this log entry. If the log is system generated, the value of this column will be "System" (Read-only)
Name	name	Name of log entry. (Read-only)
Data	max_bytes_64	Value of log entry. (Read-only)

`Log` tables are array tables that store log entries. Each row in a `Log` table is an entry.

There may be more than one `Log` table in an SP. Each of these `Log` tables must have a unique name. Only the default `Log` table, which has the name “Log”, stores System log entries. The Log table described in this section acts as the template for all log tables. Log tables are created using the `CreateLog` method. User-created log tables each have an associated row in the `LogList` table.

The `LogTo` column of the `Method` table allows the host to associate an access control association to a particular log table. All access control associations enumerated in the `Method` table at issuance shall be logged to the default log.

The `LogTo` column of the `Authority` table allows the host to associate an authority’s operations to a particular log table. All authorities at issuance shall be logged to the default log.

Logs are maintained in a cyclical manner. For efficiency, all rows in a `Log` table should be pre-allocated (that is, none of them is free). The value 0 in the `LogKind` column of a row indicates that that row has not yet been used. As log entries are added, the value in the `LogKind` column for each of those used rows changes to reflect the type of log entry added.

If dynamic row allocation is supported, the log table may have additional rows created. New rows are added at the end of the table. These rows, like the rows present at table creation, are considered allocated and have the value 0 in the `LogKind` column. The number of rows in a table may be changed by invoking the `Set` method on the `Rows` column of the `Table` table. For information on adding rows to a table, see section 5.3.4.2.1.

A `Log` table row contains a timestamp and uidrefs to and names of the authorities used to start the session in which the activity is being logged. Actual log data (the value stored to the `Data` column) depends on the `LogKind` field (see section 5.7.4.2).

All log entries for a single session will share the same unique host session number.

If the Clock Template has not been issued into the SP with the Log Template, when a new entry is created in a `Log` table the values for the `MonotonicTime`, `ExactTime`, and `TimeKind` columns shall be 0.

The TPer shall atomically add log entries to a log table if multiple read sessions are open to the SP and are affecting that log table.

5.7.2.2 LogList (Object Table)

Table 155 LogList Table Description

Column	IsIndex	Type	Description
UID		uid	uid of this LogList object (Read-only)
Name	Yes	name	Name of the associated Log table (Read-only)
CommonName	Yes	name	Name that may be shared by multiple Log tables (Read-only)
Log		table_ref	uid of the associated Log table (Read-only)
Serial		log_row_ref	Cursor for the associated Log table. The log is circular. Serial is incremented for each log entry and wraps around when it reaches the end of the Log table. Any row of the Log table that has LogKind = 0 marks that row as free and unused. (Read-only)

Column	IsIndex	Type	Description
HighSecurity		boolean	When HighSecurity is true, every log message is committed to persistent storage when received. When false, messages may be queued for later writing (some messages could potentially be lost when a TPer reset occurs).

The `LogList` table is an object table that contains exactly one row for each `Log` table, and contains information about that log.

The `LogList` row with `UID=0x00 0x00 0x0A 0x02 0x00 0x00 0x00 0x01` is automatically created on SP issuance with the name `Log`. A corresponding `Log` table is also created at Issuance. The uid of the created `Log` table is referenced in the `LogList's Log` column. This initial row will also have default values of `Serial=1` and `HighSecurity=false`.

The `Log` and `Serial` columns are protected columns and cannot be set by user action. Only the `Name`, `CommonName`, and `HighSecurity` values may be specified when a `Log` table is created, and Life Cycle indicates that only the `HighSecurity` value may be changed after a `Log` table is created (see 5.7.5 for details).

5.7.3 Methods

5.7.3.1 AddLog (Table Method)

```
LogTableUID.AddLog[
       LogEntryName : name,
       Data : max_bytes_64 ]
=>
[ Result : boolean ]
```

`AddLog` will add a log entry to the `Log` table on which the method was invoked.

Successful invocation of this method automatically sets the value of the `Local` column of the log entry to False and sets the `LogKind` column value to 9.

This call is permitted even during a Read-Only session. It is not subject to transactional abort/rollback. If multiple Read-Only sessions are open to the same SP, the TPer is required to update the shared log without corruption. Log entries from each session are guaranteed to be in their proper relative order, but no guarantee is made about the relative ordering of entries between separate sessions.

5.7.3.1.1 Fails

- If the referenced log table does not exist

5.7.3.2 CreateLog (Table Method)

```
LogListUID.CreateLog[
       NewLogTableName : name,
       HighSecurity : boolean,
       MinSize : uinteger_4,
       MaxSize = uinteger_4,
       Hintsize = uinteger_4,
       CommonName = name]
=>
[ LogListUID : uid,
LogTableUID : uid,
Rows : uinteger_4 ]
```

Successful invocation of this method creates a row in the `LogList` table with the given name and security level, and creates a corresponding `Log` table with the name given in the method invocation. The `Log` table described in 5.7.2.1 is used as the template for the new table. ACLs are set for the new row in the `LogList` table as if `CreateRow` had been used to create it, as described in 5.7.5. A row in the `Table` table is created as normal for the new log table.

The result of a successful `CreateLog` method invocation is the uid of the new `LogList` object, the uid of the new `Log` table, and the number of rows created in the new `Log` table.

5.7.3.2.1 Fails

- If a log table with the specified name already exists.
- If there isn't space in the SP for the new table.
- If metadata/support tables (i.e. Table, Column, Method, or ACE) are not all able to create all required rows to support this table.
- If TPer determines MinSize is too large.

5.7.3.3 ClearLog (Table Method)

```
LogTableUID.ClearLog[ ]
=>
[ Result : boolean ]
```

All entries in the indicated `Log` table are removed.

5.7.3.3.1 Fails

- If the referenced log table does not exist.

5.7.3.4 FlushLog (Table Method)

```
LogTableUID.FlushLog[ ]
=>
[ Result : boolean ]
```

Upon successful invocation of this method, all entries that exist only in the main memory and have not yet been committed to media are committed to the indicated `Log` table on media. When `HighSecurity` is true, `FlushLog` is implicitly invoked after any `AddLog` method invocation.

The **Result** is not generated until the persistent storage commit is complete.

5.7.3.4.1 Fails

- If the referenced log table does not exist.

5.7.4 Descriptions

Logs are cyclical. Implementation shall prevent uncontrolled logging recursion.

5.7.4.1 Types of Logging

There are two types of logging:

- **User** – User logging is the result of invocation of the `AddLog` method on a `Log` table.
- **System** – System log entries shall be stored in the default `Log` table, or the log table designated by the `LogTo` column of the `Authority` or `Method` table. System logging occurs automatically as the result of four classes of events:
  - Authentication attempts against an authority (success/failure). Logging for these events are controlled in the `Authority` table.
  - Method invocations (success/failure). Logging for these events are controlled in the `Method` table.
  - Transaction events (TransactionStart, TransactionEnd, TransactionAbort). Logging for transaction events is always to the default log table.
  - Session events (`StartSession`, `SyncSession`, `StartTrustedSession`, `SyncTrustedSession`, `CloseSession`). Logging for session events is always to the default log table.

Each Template reference section includes a description of the default logging values for methods and authorities provided to an SP by that Template.

5.7.4.2 Log Entries

Each log entry is a row in a `Log` table. Each of these rows includes columns for the Session ID, uidrefs to Session authorities, the names of those authorities, a timestamp, a monotonic counter value, a `LogKind`, a name for the log entry, and a data field that can hold up to 64 bytes of data for a log message.

The value of the data field for system entries is dependent on the value of the `LogKind` column.

The `LogKind` column shall have one of the following values:

- 0 = available
- 1 = methodFail
- 2 = methodSuccess
- 3 = authenticateFail
- 4 = authenticateSuccess
- 5 = transactOpen
- 6 = transactCommit
- 7 = transactAbort
- 8 = sessionEnd
- 9 = user
- 10 = system
- 11-23 = reserved

The structure of the system entry is as follows:

- Bytes 1-8 store the uid of the table involved in the operation. If the method invoked was an SP method, these bytes will be `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01`.
- Bytes 9 – 16 store the uid of the method involved in the operation. If there is no method (i.e. the log entry is recording a transaction-related operation), these bytes will be zeroes.
- Bytes 17 and 18 store a status code for the operation (i.e. `0x00` for Success).

5.7.4.3 Deleting a Log Table

Because a `Log` table and the `LogList` table must be kept in sync, there will be no ACL to allow the `Log` table to be deleted via the `Table` table. The LogList object's `Delete` method must be used to delete a `Log` table. Successful invocation of that method deletes the `Log` table and its associated entries in both the `LogList` table and the `Table` table. Note that Life Cycle will prevent the default `Log` table from being deleted in any other manner. For more information, see the Log Template Life Cycle section 5.7.5.

5.7.4.4 Default Logging Settings

The default logging settings associated with the Log Template methods are:

- The default logging setting for the `Delete` object method on objects in the `LogList` table is LogAlways.
- The default logging setting for the `ClearLog` method on all `Log` tables is LogAlways.

- All other methods that apply to the Log and LogList tables will be as described in the Base Template reference section (See Section 5.3.4.4).

5.7.5 Life Cycle

5.7.5.1 Log Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Log Template has the following characteristics based on the current life cycle state of that SP:

- **Issued –** At Issuance the SP will have the default Log Template-related access control settings as described in the following sections. Method invocations that occur to the Admin SP during the Issuance process are logged to the Admin SP default Log table (if the Log Template is part of the Admin SP and if logging of those method invocations is enabled).
- **Disabled** – A Log Template-enabled SP in the Disabled state shall log authority authentication attempts, session startup attempts, and all method invocation attempts (dependent on log settings in the Authority and Method tables). A Log Template-enabled SP that is in the Disabled state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.3.
- **Frozen –** A Log Template-enabled SP that is in the Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.4.
- **Issued-Disabled-Frozen** – A Log Template-enabled SP in the Disabled state shall log authority authentication attempts, session startup attempts, and all method invocation attempts (dependent on log settings in the Authority and Method tables). A Log Template-enabled SP that is in the Issued-Disabled-Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.5.

5.7.5.2 Initial Access Control Settings

The following sections enumerate the initial required access control settings for the table/method combinations provided to an SP by the Log Template. These access controls represent the pre-personalization settings of the Log Template-related table/method combinations, i.e. those that exist when the SP first enters the Issued state.

In the descriptive tables in this section, “None” indicates that the relevant ACL column of the Method table has a zeroes (Null UID reference). This indicates that access control to perform that action cannot be satisfied.

Some methods do not appear in the descriptive tables in this section for some Template tables or objects. This indicates that the method shall not be able to be invoked on that table or object, and there shall be no row in the Method table representing that access control association.

5.7.5.2.1 ACEs

In addition to the ACEs defined in the Base Template, which are defaults for all SPs, the following table defines the ACEs added for use in the life cycle of the Log Template.

Table 156 Log Template Added ACEs

UID	Name	BoolExpr	RowStart	RowEnd	ColStart	ColEnd
00 00 00 08 00 00 0A 01	LogList_Security	Admins	1	1	HighSecurity	HighSecurity

5.7.5.2.2 LogList Table/Objects Default Access Control Settings

Table 157 LogList Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateLog	Admins	Admins	Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	LogList_Security	None	Admins	Admins	Admins

Table 158 displays the Object ACLs on all `LogList` objects except for the first, which is the row created at Issuance that represents the system log.

Table 158 LogList Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Self	Self	Self	Self	Self

Table 159 displays the Object ACLs on the system `LogList` object, which is the row created at Issuance that represents the system log.

Table 159 Initial LogList Object Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Admins	Admins	Admins	Admins	Admins
Set	LogList_Security	None	Admins	Admins	Admins

5.7.5.2.3 Log Table Default Access Control Settings

Table 160 displays the Table ACLs on the default `Log` table, which is the `Log` table created at Issuance and used for system logging.

Note that in this table, when a new entry is added, there is no ACL that allows the `Set` method to be used on that entry, and no ACL that allows `AddACE` or `RemoveACE` to be used on the `Set` method. This is true for both system and user Log entries.

Table 160 Log Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
ClearLog	Admins	Admins	Admins	Admins	Admins
AddLog	Admins	Admins	Admins	Admins	Admins

5.7.6 Examples

5.7.6.1 Example LogList Table

Table 161 Example LogList Table

UID	Name	Log	Serial	HighSecurity
00 00 0A 02 00 00 00 01	Log	11	3	T

5.8 Locking Template

5.8.1 Overview

The Locking Template defines mechanisms for access control to user data, including controlling media encryption, user data encryption key management, and Read/Write lock state.

The SP on a TPer that has been issued with the Locking Template provides access control on firmware and electronics functions intrinsic to the Read/Write operations of the TPer. These include gating Read/Write functions (Read/Write locking) and the manipulation of keys for encryption of user data.

Functions enabled by the Locking Template are security-policy sensitive and should be provided only under proper access control. In a Security Subsystem Class in which issuance is not possible, the SP that incorporates the Locking Template may be activated by default in manufacturing, but only with the default access control settings as described in this document.

The reason that the Read/Write locking state and the encryption state are controlled in the same SP is that the change of the encryption key is logically, though not functionally, equivalent to the Read/Write locking state becoming invalid. When the TPer is in a Read/Write locked state, the absence of correct authentication is made functionally equivalent to the absence of a correct encryption key. The Read/Write lock and encryption circuits need common policy control so as to insure that one is not employed to attack the other. The default access control settings associated with the Locking Template are found in section 5.8.5.2.

The Locking Template also enables an SP to manage re-encryption of data. Re-encryption is the process by which User Data LBAs are transformed from 1) encrypted data using the active encryption key to encrypted data with a new encryption key, 2) clear text data to encrypted data with a new key, or 3) encrypted data to clear text data.

Re-encryption has the following basic attributes:

- This process operates as a background TPer operation. Re-encryption may operate concurrently with normal User Data Interface Commands.
- Re-encryption processes are linked to a specific LBA range. Multiple concurrent re-encryption operations are permitted up to the available TPer re-encryption resources.

Side effects may occur when writing cells of the tables of the SP that incorporates the Locking Template. These side effects include enabling Read/Write locking, enabling encryption with a certain encryption key, and initiating the re-encryption process.

5.8.1.1 Terminology

Table 162 Locking Template Terminology

Term	Definition
Global range	The entire User-Addressable LBA Range
Key Changing	The changing of a Credential reference
KeysAvailable	Condition: Host has provided enough information to enable access to Locking LBA range keys. See the KeysAvailableCfg column in the LockingInfo Table for more information.
LBA Range	A defined sub-section of the User-Addressable LBA Range
MBR Shadowing	This allows loading of preboot code that will be necessary to unlock an LBA range that starts at LBA 0 for reading and writing
Media Encryption	Inline encryption of data to media
Re-encryption	Encryption of the original cleartext media data, which may have been previously encrypted, to the media with a different key.

Term	Definition
TPer_Error_Detect	Condition: A TPer re-encryption error has been detected.
TPer_Job_Done	Condition: TPer has completed re-encryption without errors
TPer_Key_Error	Not all keys are valid
TPer_Ready	Condition: ALL the following resources & conditions required to begin or continue re-encryption are true: - All TPer resources are available, such as buffer space, re-encryption H/W & S/W resources. - Re-encryption keys are valid - TPer_Error_Detect condition is NOT detected - TPer_Reset_Stop condition is NOT detected - PAUSE_req is NOT TRUE - KeysAvailable is TRUE - NextKey is valid and the credential that it references is accessible by the TPer when any authority that has permission on the Set method for the ReadLocked or WriteLocked columns is authenticated.
TPer_Reset_Detect	Condition: A reset condition has been detected.
TPer_Reset_Stop	Condition: A reset condition is detected that does not permit the Re-encryption process to continue.
User-Addressable LBA Range	The user-accessible storage space on a storage device, where user data is stored

5.8.2 Data Structures

The Locking Template contains the following tables:

- **LockingInfo**: Information about the TPer's configuration
- **Locking**: The storage encryption and read/write locking controls covering different contiguous ranges of storage space on the TPer.
- **MBRControl** and **MBR**: For MBR shadowing, needed to handle boot environments.

5.8.2.1 LockingInfo (Array Table)

The `LockingInfo` table is an array table that consists of one row and contains information about the device configuration.

Table 163 LockingInfo Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	The UID of this table row. (Read-only)
Name	name	A manufacturer name for this feature (Read-only)
Version	uinteger_4	Manufacturer-defined version number (Read-only)
EncryptSupport	enc_supported	Supported encryption modes (Read-only)
MaxRanges	uinteger_4	Maximum number of LBA ranges, as set by manufacturer (Read-only)
MaxReEncryptions	uinteger_4	Maximum number of simultaneous re-encryptions permitted.

Column	Type	Description
		(Read-only)
KeysAvailableCfg	keys_avail_conds	This column defines which conditions are required to assert KeysAvailable.

EncryptSupport: If Media Encryption is supported, this is indicated here. The value of this column is "None" if the drive cannot support inline encryption of data to media.

MaxRanges: The value of this column represents the maximum number of different LBA ranges permitted by the manufacturer. If 0, then the only range available is the entire Global Range of the storage device.

MaxReEncryptions: represents the number of simultaneous re-encryptions permitted by the TPer. Simultaneous re-encryptions refer to the number of different LBA ranges that can be concurrently re-encrypted. A value of 0 indicates Re-encryption is NOT supported.

KeyAvailableCfg: represents additional conditions required for access to the LBA Range keys.

- None = no conditions are required
- 1 = an authority with `Set` access to `ReadLocked`/`WriteLocked` columns for this LBA range has been authenticated since the last detected reset condition

5.8.2.2 Locking (Object Table)

`Locking` table rows define encryption & re-encryption behavior for the storage device's LBA ranges. An LBA range is defined as an ordered sequence of `RangeLength` logical blocks (as appropriate to the device, typically LBAs), numbered consecutively starting at LBA `RangeStart`.

The `Locking` table always has at least one row. The first row of the `Locking` Table always represents the entire User-Addressable LBA Range. This row shall have a UID column value of `00 00 08 02 00 00 00 01`. This row cannot be deleted.

Additional rows shall comply with the following rules:

- Each additional row in this table represents a contiguous "subdivision" of the entire User-Addressable LBA Range.
- The number of rows in this table must not exceed the value of the `Locking_Info` table's `MaxRanges` column + 1. If `MaxRanges` = 0, this is and can only be a one row table.
- New rows of the `Locking` table are created using the `CreateRow` method. A valid `CreateRow` method requires the following attributes:
  - Values for the `RangeStart` and `RangeLength` columns shall be specified.
  - The specified `RangeStart` and `RangeLength` values shall not overlap any other row but the first row. NOTE: first row is the row that represents the entire user addressable LBA range. NOTE: TPer checks LBA range before creating a row. An overlapping request results in the `CreateRow` method returning an error.

Table 164 Locking Table Description

Column	Type	Description
UID	uid	Unique identifier for this Locking object (Read-only)
Name	name	Name (Read-only pre-personalization only)
CommonName	name	Name that may be shared across multiple Locking objects.

Column	Type	Description
RangeStart	uinteger_8	LBA starting address, Ignored if MaxRanges = 0 (Read-only)
RangeLength	uinteger_8	Quantity of LBAs, including RangeStart LBA. (Read-only)
ReadLockEnabled	boolean_def_false	Identifies whether ReadLocking is enabled for this range.
WriteLockEnabled	boolean_def_false	Identifies whether WriteLocking is enabled for this range
ReadLocked	boolean	The current read locking state
WriteLocked	boolean	The current write locking state
LockOnReset	reset_types	Identifies the LBA range's storage-related locking behavior, *dependent on reset type*. Note that both Read and Write Locking behavior on reset are controlled by this value. An empty set means locking does not occur on any reset.
ActiveKey	cred_object_uidref	Points to the present encryption key for this LBA range.
NextKey	cred_object_uidref	Points to the next encryption key for this LBA range.
ReEncryptState	reencrypt_state	This is the present Re-encryption State for this LBA range. ReEncryptState reports the TPer's response to re-encrypt requests. (Read-only)
ReEncryptRequest	reencrypt_request	This value represents a re-encryption request.
AdvKeyMode	adv_key_mode	This value defines when NextKey is moved to ActiveKey
VerifyMode	verify_mode	Defines verification operation after sector is written with new encryption key.
ContOnReset	reset_types	Defines the re-encryption behavior after a reset condition is detected.
LastReEncryptLBA	bytes_max_32	Represents the last good re-encrypted LBA. It is only valid when the ReEncryptState column value is ACTIVE, COMPLETED, PENDING or PAUSED. (Read-only)
LastReEncStat	last_reenc_stat	Last attempted Re-encryption step. Only valid when the ReEncryptState column value is COMPLETED, PENDING, or PAUSED state. (Read-only)
GeneralStatus	gen_status	Reason for arriving at PAUSED or PENDING state. (Read-only)

The first row of the `Locking` table (the Global Range) shall have a `Name` column value of **Global_Range** and a `UID` column value of `0x00 0x00 0x08 0x02 0x00 0x00 0x00 0x01`.

The first row of the Locking table (the Global Range) shall have a `CommonName` column value of **Locking**.

RangeStart: this column value defines the starting LBA value for this range.

RangeLength: This column value defines the quantity of contiguous LBAs for this LBA range. If RangeLength = 0 and the RangeStart = 0 then this pair specifies the storage device's global range. The Global Range shall only be specified in a single row. This field is ignored if MaxRanges = 0.

ReadLockEnabled: The value of this column determines whether or not the read-locking feature is enabled for this scope. Enabled in this sense means whether or not the ReadLocked column value is meaningful for this range. If the value of the ReadLockEnabled column is False, the read-locking feature is disabled, and the value of the ReadLocked column is disregarded. If the value of ReadLockEnabled column is True, the read-locking feature is enabled. The value of the ReadLocked column identifies the current read locking state.

WriteLockEnabled: The value of this column determines whether or not the write-locking feature is enabled for this scope. Enabled in this sense means whether or not the WriteLocked column value is meaningful for this range. If the value of the WriteLockEnabled column is False, the write-locking feature is disabled, and the value of the WriteLocked column is ignored. If the value of WriteLockEnabled column is True, the write-locking feature is enabled. The value of the WriteLocked column identifies the current write locking state.

ReadLocked: The value of this column identifies the current read lock state for the associated LBA Range. If this value is True, the range is read-locked. If this value is False, the range is read-unlocked.

The Set method may be invoked by the host to change the value of this column and alter the read-lock state. Setting the column value to 1 read locks the range. Setting the column value to 0 read unlocks the range.

WriteLocked: The value of this column identifies the current write lock state for the associated LBA Range. If this value is True, the range is write-locked. If this value is False, the range is write-unlocked.

The Set method may be invoked by the host to change the value of this column and alter the write lock state. Setting the column value to 1 write locks the range. Setting the column value to 0 write unlocks the range.

LockOnReset: This value defines the locking behavior of this LBA range at reset, dependent on reset type. The values enumerated in this column identify the reset types that cause the values of the ReadLocked and WriteLocked columns of the Locking table to be set to True (assuming the associated *LockEnabled values are True for those columns).

The default value of this column is 0.

The Global Range's LockOnReset value defines global TPer behavior. Row 2-n override the Global Range's behavior, unless otherwise specified in an SSC.

ActiveKey: This field points to this LBA range's media encryption key. If the value of this column is 00's then data in this range is stored in plaintext.

The following rules define how and when the ActiveKey column value is modified:

- Host Application directly writes ActiveKey column value
- When the ReEncryptState column value is COMPLETED and the Host sets the ReEncryptRequest column value to ADVKey_req, the TPer moves the NextKey column value to the ActiveKey column (setting the NextKey column to a Null UID reference).
- When the ReEncryptState column value is ACTIVE AND AdvKeyMode = 1 AND TPer_Job_Done condition is detected, the TPer moves the NextKey column value to the ActiveKey column (setting the NextKey column to a Null UID reference).
- When ReEncryptState value is PAUSED AND the Host sets ReEncryptRequest to ADVKey_req, the TPer moves the NextKey column value to the ActiveKey column (setting the NextKey column to a Null UID reference).

NextKey: This column identifies the LBA range's next media encryption key. This value and the referenced credential object shall be writable when the value of the ReEncryptState column is IDLE only. Otherwise, attempts to invoke an of the Set, Delete, or DeleteRow methods on the associated credential object shall return an error.

User Data shall be returned to clear text when the key value stored at NextKey is zeroes AND Re-encryption has been requested

ReEncryptState: (read only) the value of this column identifies the currently applicable Re-encryption state (see Figure 22). The value in the column identifies the TPer's response to the host's requests in the ReencryptRequest column.

Reset configuration (ContOnReset) and a detected reset condition define the reported ReEncryptState and PauseStatus values.

When the ReEncryptState column value is:

- 1 = IDLE: re-encryption is not active for this LBA range.
- 2 = PENDING: This LBA Range's re-encryption process is waiting to start or continue re-encryption.
- 3 = ACTIVE: This LBA Range's re-encryption process is executing
- 4 = COMPLETED: This LBA Range's re-encryption process has completed without errors
- 5 = PAUSED: This LBA Range's re-encryption has temporarily halted.

ReEncryptRequest: A host application requests a re-encryption operation by writing to this column. Successful invocation of the Get method on this column shall always return "0".

Only state transitions described in Figure 22 shall be valid.

If a Set method invocation attempts to set a value to the ReEncryptRequest column that is not valid for the current ReEncryptState column value, then this Set method invocation shall return an error.

- 1 = START_req: Host requests a new re-encryption process. Only accepted when the ReEncryptState column value is IDLE. The TPer changes the value of the ReEncryptState column to PENDING.
- 2 = ADVKEY_req: Host requests TPer to change the ReEncryptState column value to IDLE AND move the value of the NextKey column to the ActiveKey column (this move also sets the value of the NextKey column to a Null UID reference). This request is only valid when the ReEncryptState column is COMPLETED or PAUSED.
- 3 = RETIDLE_req: Host requests a return to the IDLE state WITHOUT moving keys. This request provides the host application control over re-encryption resources and background activity.
- 4 = CONT_req: Host requests the re-encryption process transition ReEncryptState=PAUSED state to ReEncryptState=PENDING. Re-encryption begins at (LastReEncryptLBA + 1). Invocation of the Set method to perform this operation shall only succeed if the current ReEncryptState column value is PAUSED.
- 5 = PAUSE_req: Host requests the quiescing of the re-encryption process. Invocation of the Set method to perform this operation shall only succeed if the current ReEncryptState column value is ACTIVE or PENDING.

AdvKeyMode: This value defines when 1) the value of the NextKey column moves to the ActiveKey column AND 2) the ReEncryptState value changes.

- When AdvKeyMode = 0 AND TPer_Job_Done condition is detected AND ReencryptState is ACTIVE, TPer changes the ReEncryptState value to COMPLETED.
- When AdvKeyMode = 1 AND TPer_Job_Done condition is detected AND the value of the ReencryptState column is ACTIVE, the TPer changes the ReEncryptState column value to

IDLE. In addition, the TPer changes the value of the ActiveKey column to be the value of the NextKey column, and then sets the value of the NextKey column to a Null UID reference.

- When AdvKeyMode = 0 AND Reencryptstate is COMPLETED AND AdvKey_req is True, the TPer changes the ReEncryptState column value to IDLE AND NextKey becomes ActiveKey.

VerifyMode: This column value defines the verification requirement during re-encryption. When True, a Read Verify shall be performed on the re-encrypted LBA before the LBA is considered good.

ContOnReset: This column value is a list of reset conditions. This value defines how a re-encryption process reacts to reset conditions.

- A value of Null means the TPer_Reset_Stop condition is set for any reset condition. The ReEncryptState value is set to PAUSED.
- For each listed reset entry, the re-encryption process may continue after the associated reset is detected.

LastReEncryptLBA; This column value defines the last good re-encrypted LBA for this region. This field is only valid when the ReEncryptState is ACTIVE, COMPLETED, PENDING, or PAUSED. Typically, when the ReencryptState is ACTIVE, this value is updated periodically. In COMPLETED, PENDING, or PAUSED this value shall be valid. When no LBA has been successfully been re-encrypted, the value is 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF.

LastReEncStat: (read only) this column value defines the last good re-encryption read-modify-write-verify sequence. This column value is only valid when the ReencryptState is COMPLETED, PENDING or PAUSED. When the LastReEncStat value is anything other than SUCCESS, the value of LastReEncryptLBA+1 is the LBA in error. Valid values are:

- 0 = Success:
- 1 = Read error: unable to read original LBA from media
- 2 = Write error: unable to write new LBA to media
- 3 = Verify error: unable to verify new LBA on media

GeneralStatus: This field defines why the Re-encryption operation arrived at the PAUSED or PENDING state. Status includes:

Values 0-31 are valid for the PAUSED state, value 32-63 are valid for the PENDING state.

- 1 = pending_tper_error: last ReEncryptState value was PENDING AND a TPer_Error_Detect condition was detected
- 2 = active_tper_error: last ReEncryptState value was ACTIVE AND a TPer_Error_Detect condition was detected
- 3 = active_pause_requested: last ReEncryptState value was ACTIVE AND PAUSE_req was detected
- 4 = pend_pause_requested: last ReEncryptState value was PENDING AND a PAUSE_req value was detected
- 5 = pend_reset_stop_detect: A reset condition AND its associated ContOnReset configuration does not allow re-encryption to continue AND last state was PENDING
- 6 = key_error: ReEncryptState value was PENDING AND valid keys were not found in any C_* table OR insufficient access control granted for reading C_* table.
- 32 = wait_AvailableKeys: keys are not available
- 33 = wait_for_TPer_resources: TPer_Ready condition is not True
- 34 = active_reset_stop_detect: A reset condition AND its associated ContOnReset configuration does not allow re-encryption to continue AND last ReEncryptState value was ACTIVE

5.8.2.3 MBRControl (Array Table)

Table 165 MBR_Control Table Description

Column	Type	Description
RowNumber	uinteger_4	This is the row number for this row of this array table, as assigned and maintained by the TPer. (Read-only)
UID	uid	The UID of this table row. (Read-only)
Enable	boolean	Enable Master Boot Record Shadowing
Done	boolean	Indicates if Master Boot Record Shadowing is done.

This table shall have only one row, with `UID=0x00 0x00 0x08 0x03 0x00 0x00 0x00 0x01`.

If `Enable` is set to True, then on the next TPer power cycle or hardware reset, the TPer shall respond to LBA requests with host-requested values from the `MBR` table, and shall remap further LBA requests to blocks in the `MBR` table until `Done` is set to True. The `Done` column is set to False on every power cycle or hardware reset. `Done` is effective only on the next power cycle or reset.

The `Done` column is set to True either by the code in the `MBR` table or by a host application. Access control must be satisfied in order to set the value of the `Done` column.

After a power cycle or hardware reset, until the host sets the `MBR_Control` table's `Done` column to True, LBA requests made by the host shall only be fulfillable by values from the `MBR` table.

The values in the `MBR` table shall only be modifiable by properly authenticated `Set` method invocations, even during the boot process. After a power cycle or hardware reset, until the `MBR_Control` table's `Done` column is set to True, attempts to write MBR space via traditional interface commands abort and return an error.

The LUN associated with the MBR is the boot LUN. A TPer that has multiple LUNs may have multiple TPers – each TPer shall be associated with a different LUN. Every LUN on a device is not required to have a TPer, but LUNs that support the TCG commands and functionality shall have a TPer. A TPer shall only be associated with exactly one LUN.

5.8.2.4 MBR (Byte Table)

Table 166 MBR Table Description

Column	Type	Description
Data	bytes {x}	Bytes to be loaded by MBR Request.

The size of the table (defined as “bytes {x}” in the Type column of the `MBR` descriptive table) is Security Subsystem Class (SSC) specific, and is to be specified in the description of each SSC that supports the use of this functionality. The size of the `MBR` table shall be retrievable from the Admin SP’s `Table` table.

Once in MBR mode, the storage capacity reported by the TPer may not be the capacity of the user-addressable LBA space.

Media encryption does not apply to the code stored in the `MBR` table, since this code is part of the secure area and media encryption applies to the user LBA ranges defined in the `Locking` table.

Errors during boot execution of the code in the `MBR` table are handled through regular boot error reporting mechanisms.

Interface-specific errors and SMART logging behaviors that occur during execution of the `MBR` table shall be vendor-specific.

5.8.3 Methods

5.8.3.1 GetPackage Method (Object Method)

```
CredentialObjectUID.GetPackage [
       Purpose : package_purpose,
       WrappingKey : Authority_ref,
       HashType : hash_protocol,
       Date = date,
       Log = max_bytes_64
=>
[Result : package]
```

The purpose of this method is to retrieve key material from a credential table in a secure manner. The return result is a "package", with the following format, where the plus sign (+) is used to indicate concatenation:

- Package = Credential Table Name + WrappingKey(Key Material) + Purpose + Date + Log + signed(hash(all before))
  - The "Credential Table Name" is the name of the credential table containing the credential object upon which this method was invoked.
  - The "WrappingKey(Key Material)" is the key material from the invoking credential object, encrypted with the credential referenced form the **WrappingKey** parameter's authority object. This credential may be a symmetric key or the public key of a public/private key pair. When retrieving the key material from credentials that store key information in multiple columns, columns that are empty shall return as 0 for uinteger type columns and 00s for bytes type columns.
  - The "Purpose" is the **Purpose** parameter from the method invocation.
  - The "Date" is the **Date** parameter supplied to the method. This value may be omitted.
  - The "Log" is the **Log** parameter supplied to the method. This value may be omitted.
  - The "signed(hash(all before))" is a hash of the package contents (except this one), signed by the WrappingKey authority's credential. The hash protocol used to create the hash is the hash protocol specified in the HashType parameter. This is the signature of a private key for referenced public key credentials, or HMAC for referenced symmetric key credentials.

The package_purpose type is an enum value with {1=Issuance, 2 = Key Wrapping, 3 = Backup, 4-24 = reserved}.

5.8.3.1.1 Fails

- If the object does not exist

5.8.3.2 SetPackage Method (Object Method)

```
CredentialObjectUID.SetPackage [
       Value : package,
       WrappingKey : Authority_ref,
       HashType : hash_protocol
=>
[ Result : boolean ]
```

The `SetPackage` method is used to set the key material columns of a credential with key material that is sent securely to the TPer.

The `SetPackage` method takes a value that shall be the result of a successful `GetPackage` method invocation, and an authority object uidref to the authority that references the credential that can be used to decrypt the encrypted key contents of the package.

The TPer decrypts the key material using the credential referenced by the **WrappingKey** authority and verifies the signed hash using that key and the hash protocol identified in the method invocation. The TPer then verifies that the Credential Table Name in the package matches the Credential Table Name of the Destination credential object parameter. The TPer then sets columns of Destination credential object with the decrypted key material from the package.

The Log portion of the unwrapped package will cause that log entry to be made to the default `Log` table, along with the Date portion. If the Log Template has not been issued into the SP, then the Log and Date data are disregarded.

5.8.3.2.1 Fails

- If the object does not exist
- If the Destination object does not exist
- If the WrappingKey authority does not exist or does not reference a valid credential.
- If Credential Table Name from the package does not match the name of the Destination credential's containing table
- If the signed hash cannot be verified

5.8.4 Description

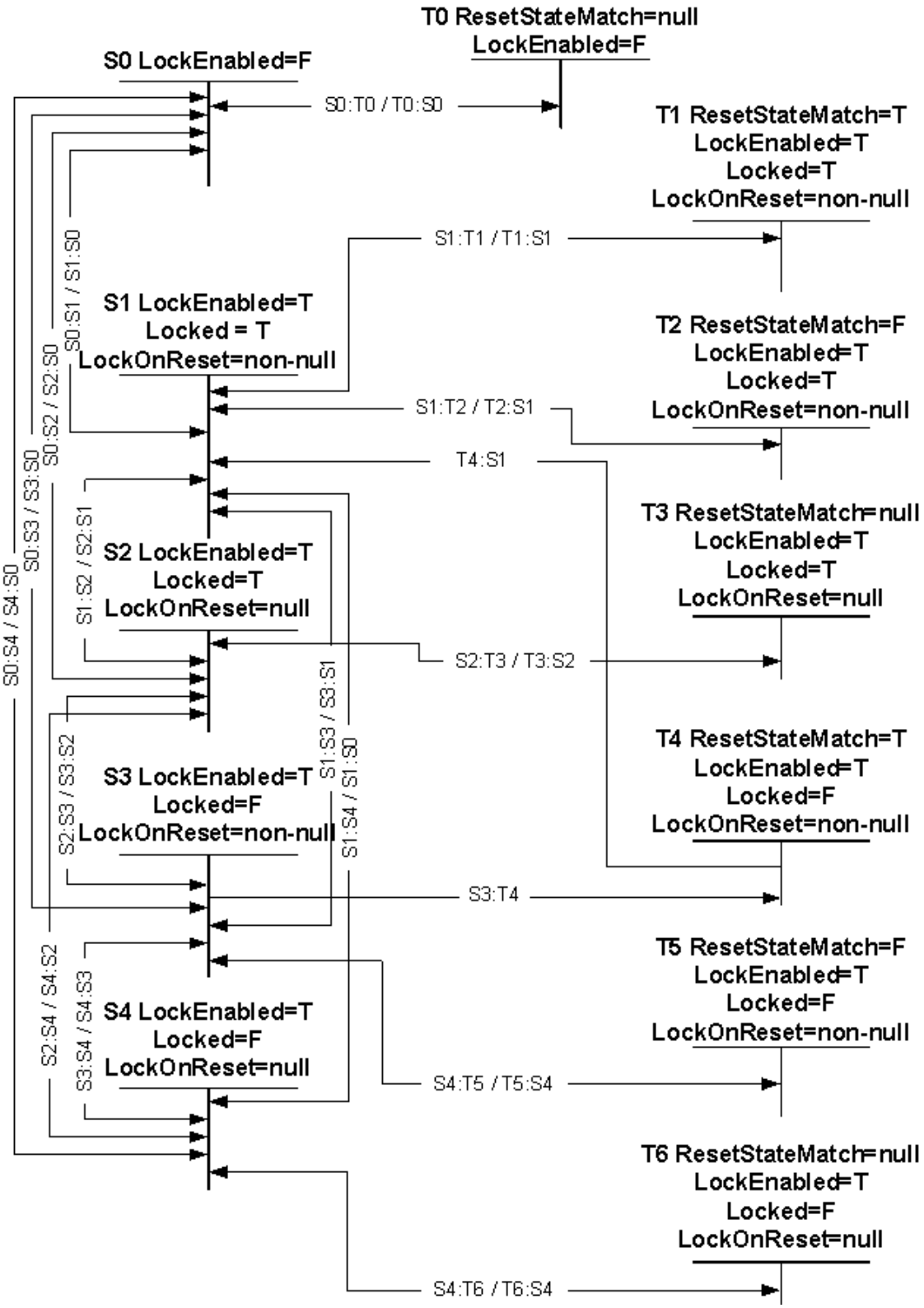
5.8.4.1 Locking State Descriptions

Figure 21 describes the states and state transitions for the locking types (read lock and write lock). For simplicity, the diagram and the accompanying textual information describe the operation of one of the locking types (read or write locking), but the behavior of the other locking type can be seen by applying the diagram separately (or simultaneously) to each type.

Note that the reset behavior of both read and write locking for each locking object is controlled at the same point, by a single column in the `Locking` table, called `LockOnReset`.

When a reset is described in these state transitions, "reset" is used generically to refer to qualifying resets, as determined by the value of the `LockOnReset` column and the reset behavior associated with particular resets as determined by the appropriate interface-specific description of that reset. Interface-specific reset definitions are defined in the appropriate TCG specification for that interface type.

Figure 21 Locking State Diagram



5.8.4.1.1 State Descriptions

This section describes the states that are used in Figure 21 , and the column values that each state represents.

S0 LockEnabled=F

This describes the state where the TPer's Locking feature is turned off. Locking is not possible. The `Locked` column and `LockOnReset` column values are disregarded.

S1 LockEnabled=T/Locked=T/LockOnReset=non-null

This describes the state where the TPer's Locking feature is turned on. Locking is possible. The `Locked` state is currently True, indicating that the range is locked. `LockOnReset` is non-null, indicating that, upon any of the listed reset events, the range will lock.

S2 LockEnabled=T/Locked=T/LockOnReset=null

This describes the state where the TPer's Locking feature is turned on. Locking is possible. The Locked state is currently True, indicating that the range is locked. `LockOnReset` is "False" (null set), indicating that reset events do not cause the range to lock. The range will maintain current locking state (the value of the `Locked` column remains the same, True) through all resets.

S3 LockEnabled=T/Locked=F/ LockOnReset=non-null

This describes the state where the TPer's Locking feature is turned on. Locking is possible. The `Locked` state is currently False, indicating that the range is not locked. `LockOnReset` is "True" (non-null set), indicating that the listed reset events cause the range to lock.

S4 LockEnabled=T/Locked=F/ LockOnReset=null

This describes the state where the TPer's Locking feature is turned on. Locking is possible. The current Locked state is False, indicating that the range is not locked. `LockOnReset` is "False" (null set), indicating that reset events do not cause the range to lock. The range will maintain current locking state (False in this case) through all reset events.

T0 ResetStateMatch=null/LockEnabled=F

This is the transition state where a reset is occurring and the Locking feature is disabled.

T1 ResetStateMatch=T/LockEnabled=T/Locked=T/ LockOnReset=non-null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is locked, and the `LockOnReset` value applies to the currently occurring reset state.

T2 ResetStateMatch=F/LockEnabled=T/Locked=T/ LockOnReset=non-null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is locked, and the `LockOnReset` value does not apply to the currently occurring reset state. This state is functionally equivalent to T3.

T3 ResetStateMatch=null /LockEnabled=T/Locked=T/ LockOnReset=null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is locked, and the `LockOnReset` value is null. This state is functionally equivalent to T2.

T4 ResetStateMatch=T/LockEnabled=T/Locked=F/ LockOnReset=non-null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is not locked, and the `LockOnReset` value applies to the currently occurring reset state.

T5 ResetStateMatch=F/LockEnabled=T/Locked=F/ LockOnReset=non-null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is not locked, and the `LockOnReset` value value does not apply to the currently occurring reset state. This state is functionally equivalent to T6.

T6 ResetStateMatch=null /LockEnabled=T/Locked=F/ LockOnReset=null

This describes a transition state where a reset is occurring and the range had the accompanying attributes - the locking feature is turned on, the range is not locked, and the `LockOnReset` value is null. This state is functionally equivalent to T5.

5.8.4.1.2 State Change Descriptions

This section describes the state changes depicted in the picture. In parentheses next to each state transition identifier are the values that change to cause that transition. "Reset" indicates that a reset occurs to cause the state change. "ResetStateMatch" is used to indicate if a reset event type that occurred is applicable or matches the `LockOnReset` column value.

S0:T0 (Reset)

This state change occurs as the result of some device reset event. The locking range with `LockingEnabled`=F exits the reset state into its previous state.

S0:S1 (LockEnabled=T, Locked=T, LockOnReset=non-null)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to True, the `Locked` column value to True, and the `LockOnReset` column value to non-null.

S0:S2 (LockEnabled=T, Locked=T, LockOnReset=null)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to True, the `Locked` column value to True, and the `LockOnReset` column value to null.

S0:S3 (LockEnabled=T, Locked=F, LockOnReset=non-null)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to True, the `Locked` column value to False, and the `LockOnReset` column value to non-null.

S0:S4 (LockEnabled=T, Locked=F, LockOnReset=null)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to True, the `Locked` column value to False, and the `LockOnReset` column value to null.

S1:S0 (LockEnabled=F)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to False from True.

S1:S2 (LockOnReset=null)

This state change occurs when the host invokes the `Set` method to change the range's `LockOnReset` column value to null from non-null. The value of the `LockEnabled` column is still True, and the value of the corresponding `Locked` column is still True.

S1:S3 (Locked=F)

This state change occurs when the host invokes the `Set` method to change the range's `Locked` column to False from True. The value of the corresponding `LockEnabled` column is still True, and the value of the `LockOnReset` column is still non-null.

S1:S4 (Locked=F, LockOnReset=null)

This state change occurs when the host invokes the Set method to change the range's Locked column to False from True, and the value of the LockOnReset column from null to non-null. The value of the corresponding LockEnabled column is still True.

S1:T1 (Reset, ResetStateMatch=T)

This state change occurs as the result of some device reset event, where the reset type matches the value defined in the LockOnReset column.

S1:T2 (Reset, ResetStatematch=F)

This state change occurs as the result of some device reset event, where the reset type does not match the value defined in the LockOnReset column.

S2:S0 (LockEnabled=F)

This state change occurs when the host invokes the Set method to change the range's LockEnabled column value to False from True.

S2:S1 (LockOnReset=non-null)

This state change occurs when the host invokes the Set method to change the range's LockOnReset column value to non-null from null. The value of the LockEnabled column remains True, and the value of the corresponding Locked column remains True.

S2:S3 (Locked=F, LockOnReset=non-null)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True, and to change the range's LockOnReset column value to non-null from null. The value of the LockEnabled column remains True.

S2:S4 (Locked=F)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True. The value of the corresponding LockEnabled column remains True, and the value of the LockOnReset column remains null.

S2:T3 (Reset)

This state change occurs as the result of some device reset event. The range with a LockOnReset column value of null and other column values of the S2 state exits the T3 state back into the S2 state.

S3:S0 (LockEnabled=F)

This state change occurs when the host invokes the Set method to change the range's LockEnabled column value to False from True.

S3:S1 (Locked=F)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True. The value of the corresponding LockEnabled column remains True, and the value of the LockOnReset column remains non-null.

S3:S2 (Locked=T, LockOnReset=null)

This state change occurs when the host invokes the Set method to change the range's Locked column value to True from False, and to change the range's LockOnReset column value to null from non-null. The value of the LockEnabled column remains True.

S3:S4 (LockOnReset=null)

This state change occurs when the host invokes the Set method to change the range's LockOnReset column value to null from non-null. The value of the LockEnabled column remains True, and the value of the corresponding Locked column remains False.

S3:T4 (Reset, ResetStateMatch=T)

This state change occurs as the result of some device reset event, where the reset type matches the value defined in the LockOnReset column.

S3:T5 (Reset, ResetStateMatch=F)

This state change occurs as the result of some device reset event, where the reset type does not match the value defined in the LockOnReset column.

S4:S0 (LockEnabled=F)

This state change occurs when the host invokes the Set method to change the range's LockEnabled column value to False from True.

S4:S1 (Locked=T, LockOnReset=non-null)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True, and the value of the LockOnReset column from null to non-null. The value of the corresponding LockEnabled column remains True.

S4:S2 (Locked=T)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True. The value of the LockEnabled column remains True, and the value of the LockOnReset column remains null.

S4:S3 (LockOnReset=non-null)

This state change occurs when the host invokes the Set method to change the range's LockOnReset column from null to non-null. The value of the LockEnabled column remains True, and the value of the corresponding Locked column remains False.

S4:T6 (Reset)

This state change occurs as the result of some device reset event. The range with a LockOnReset column value of null and other column values of the S4 state exits the T6 state back into the S4 state.

T0:S0 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. This state change represents behavior of a range for which LockEnabled is False.

T1:S1 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the reset event matched that specified in the range's LockOnReset column. This causes the device to enter the S1 state upon reset recovery, with a LockEnabled column value of True, a corresponding Locked column value of True, and the same LockOnReset column value as existed immediately preceding entry to T1.

T2:S1 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the reset event did not match that specified in the range's LockOnReset column. This causes the device to recover from the reset with the same LockEnabled, Locked, and LockOnReset column values that existed previous to entry to T2.

T3:S2 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the LockOnReset column value of null causes the range to recover from the reset with the same LockEnabled, Locked, and LockOnReset column values that existed previous to entry to T3.

T4:S1 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the reset event matched that specified in the range's `LockOnReset` column. This causes the device to enter the S1 state upon reset recovery, with a `LockEnabled` column value of True, a corresponding `Locked` column value of True, and the same `LockOnReset` column value as existed immediately preceding entry to T4.

T5:S3 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the reset event did not match that specified in the range's `LockOnReset` column. This causes the device to recover from the reset with the same `LockEnabled`, `Locked`, and `LockOnReset` column values that existed previous to entry to T5.

T6:S4 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the `LockOnReset` column value of null causes the range to recover from the reset with the same LockEnabled, `Locked`, and `LockOnReset` column values that existed previous to entry to T6.

5.8.4.2 Re-encryption Overview

The host has the following re-encryption responsibilities:

- Configures re-encryption options
- Initiates re-encryption operations
- Manages error recovery strategy when TPer detects errors. The host defines the next re-encryption state
- Optionally, acknowledge re-encryption completion

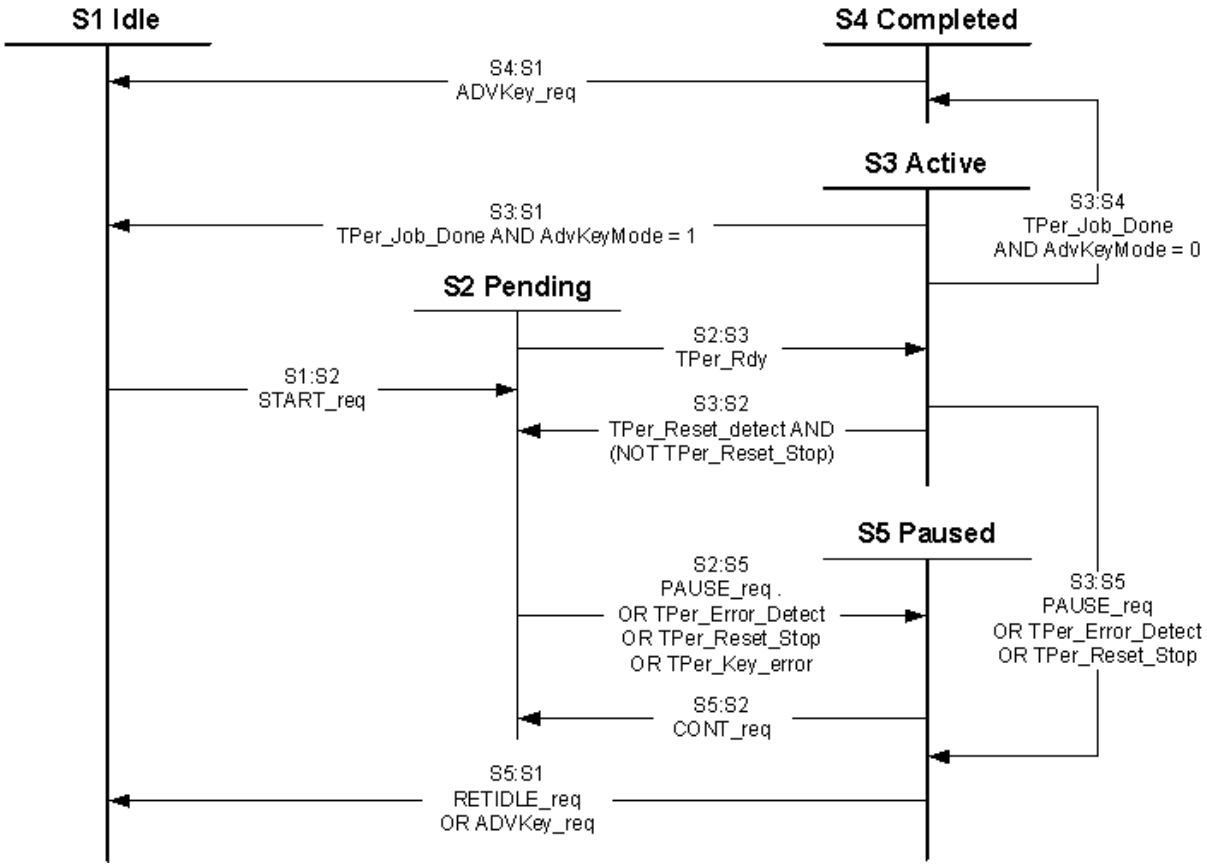
The TPer has the following re-encryption responsibilities:

- Maintain persistent re-encryption state and status information across power cycles.
- Quiesce and report re-encryption state and status
- Detect re-encryption errors, completion and reset conditions

Re-encryption and normal Read/Write commands are concurrent activities. Re-encryption is a TPer background task. As such, synchronization between normal Read/Write command processing and background re-encryption processing is required. The means by which this synchronization is accomplished is implementation dependent. However, the normal Storage Device firmware requires a way to view the re-encryption process so that the proper encryption keys are selected for User_data Read/Write commands.

5.8.4.3 Re-encryption State Descriptions

Figure 22 LBA Range Re-encryption State Diagram



S1: IDLE: Re-encryption is idle in this state. No re-encryption is executing for this LBA range.

Transition S1:S2 `START_req` has been detected.

S2: PENDING: This LBA Range's re-encryption process is waiting to start or continue re-encryption

Transition S2:S3 `TPer_Ready` condition is met.

Transition S2:S5 `PAUSE_req` has been detected OR TPer detected error condition is met

S3: ACTIVE: This LBA Range's re-encryption process is executing
Transition S3:S4 `TPer_Job_Done` condition is met AND `AdvKeyMode = 0`.

Transition S3:S2 `TPer_Reset_detect` condition is met AND `TPer_Reset_Stop` is not met.

Transition S3:S5 A `TPer_Error_Detect` condition is met OR `TPer_Reset_Stop` condition is met OR `PAUSE_req` has been detected

Transition S3:S1 `Tper_Job_Done` condition is met AND `ADVKeyMode = 1` AND re-encryption has completed

S4: COMPLETED: This LBA Range's re-encryption process has completed without errors.

Transition S4:S1 `ADVKey_req` has been detected.

S5: PAUSED: This LBA Range's re-encryption has temporarily halted. The re-encryption process has been quiesced and awaiting host intervention.

Transition S5:S2 `CONT_req` has been detected

Transition S5:S1 `ADVKey_req` OR `RETIDLE_req` has been detected

5.8.4.4 Default Logging Settings

The default logging settings associated with the Locking Template methods are:

- The default logging setting for the `Delete` object method, the `DeleteRow` table method, and the `Set` method on objects in the `Locking` table shall be LogAlways.
- The default logging setting for the `Set` method on the `Locking_Info` table, the `MBR_Control` table, and the `MBR` table shall be LogAlways.
- All other methods that apply to the Locking Template-related tables shall be as described in the Base Template reference section on Default Logging Settings (See 5.3.4.4).

5.8.5 Life Cycle

5.8.5.1 Locking Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Locking Template has the following characteristics based on the current life cycle state of that SP:

- **Issued** – At Issuance the SP will have the default Locking Template-related access control settings described in the following sections.
- **Disabled –** A Locking Template-enabled SP that is in the Disabled state shall not be able to perform any use-invoked SP operations, with the exceptions noted in section 4.4.3. This may result in boot-up failure (if MBR Shadowing is enabled), or the inability to lock or unlock certain LBA ranges for reading and/or writing.
- **Frozen** – A Locking Template-enabled SP that is in the Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.4. This may result in boot-up failure (if MBR Shadowing is enabled), or the inability to lock or unlock certain LBA ranges for reading and/or writing.
- **Issued-Disabled-Frozen** – A Locking Template-enabled SP that is in the Issued-Disabled-Frozen state shall not be able to perform any user-invoked SP operations, with the exceptions noted in section 4.4.5. This may result in boot-up failure (if MBR Shadowing is enabled), or the inability to lock or unlock certain LBA ranges for reading and/or writing.

5.8.5.2 Initial Access Control Settings

The following sections enumerate the initial required access control settings for the table/method combinations provided to an SP by the Locking Template. These access controls represent the pre-personalization settings of the Locking Template-related table/method combinations, i.e. those when the SP enters the Issued state.

In the descriptive tables in this section, "None" indicates that the relevant ACL column of the `Method` table has a Null UID reference (zeroes). This indicates that access control to perform that action cannot be satisfied.

Some methods do not appear in the descriptive tables in this section for some Template tables or objects. This indicates that the method shall not be able to be invoked on that table or object, and there shall be no row in the `Method` table representing that access control association.

5.8.5.2.1 ACEs

In addition to the ACEs defined in the Base Template, which are defaults for all SPs, the following table defines the ACEs added for use in the life cycle of the Locking Template.

Table 167 Locking Template Added ACEs

UID	Name	BoolExpr	RowStart	RowEnd	ColStart	ColEnd
00 00 00 08 00 00 08 01	LockingInfo_1	Admins			KeysAvailableCfg	KeysAvailableCfg
00 00 00 08 00 00 08 02	Locking_1	Makers			Name	RangeLength
00 00 00 08 00 00 08 03	Locking_2	Admins			ReadLockEnabled	NextKey
00 00 00 08 00 00 08 04	Locking_3	Admins			ReEncryptRequest	ContOnReset

5.8.5.2.2 Locking_Info Table Default Access Control Settings

Table 168 LockingInfo Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
DeleteRow	None	None	None	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	LockingInfo_1	None	Admins	Admins	Admins

5.8.5.2.3 Locking Table/Objects Default Access Control Settings

Table 169 Locking Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
CreateRow	Admins	Admins	Admins	Admins	Admins
DeleteRow	PI_Admins	None	Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	Locking_1, Locking_2, Locking_3	None	Admins	Admins	Admins

Table 170 displays the Object ACLs on all `Locking` objects except for the first, which is the row created at Issuance that represents the Global Range (the TPer's entire storage area).

Table 170 Locking Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Delete	Self	Self	Self	Self	Self
Get	Self	Self	Self	Self	Self
Set	Locking_2, Locking_3	Self	Self	Self	Self

5.8.5.2.4 MBR_Control Table Default Access Control Settings

This section defines the default access control settings for the `MBR_Control` table.

Table 171 MBR_Control Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
DeleteRow	Admins	Admins	Admins	Admins	Admins
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	Admins	Admins	Admins	Admins	Admins

5.8.5.2.5 MBR Table Default Access Control Settings

This section defines the default access control settings for the `MBR` table.

Table 172 MBR Table Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
Get	Admins	Admins	Admins	Admins	Admins
Next	Admins	Admins	Admins	Admins	Admins
Set	Admins	Admins	Admins	Admins	Admins

5.8.5.2.6 C_RSA_* Objects Access Control Settings – Locking Template Methods

All objects created in C_RSA_* tables in an SP into which the Locking Template has been issued have the access control settings associated with Locking Template methods as defined in Table 173.

Table 173 C_RSA_* Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
GetPackage	Self	Self	Self	Self	Self
SetPackage	Self	Self	Self	Self	Self

5.8.5.2.7 C_EC_* Objects Access Control Settings – Locking Template Methods

All objects created in C_EC_* tables in an SP into which the Locking Template has been issued have the access control settings associated with Locking Template methods as defined in Table 174.

Table 174 C_EC_* Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
GetPackage	Self	Self	Self	Self	Self
SetPackage	Self	Self	Self	Self	Self

5.8.5.2.8 C_AES_* Objects Access Control Settings – Locking Template Methods

All objects created in C_AES_* tables in an SP into which the Locking Template has been issued have the access control settings associated with Locking Template methods as defined in Table 175.

Table 175 C_AES_* Objects Default Access Control Settings

Method	ACL	AddACE	RemoveACE	GetACL	DeleteMethod
GetPackage	Self	Self	Self	Self	Self
SetPackage	Self	Self	Self	Self	Self

5.8.6 Examples

5.8.6.1 Re-encryption Functionality Examples

5.8.6.1.1 Clear Text to Encrypt Key 1

The following sequence applies to a User Data LBA region that is currently not encrypted. Preceding this sequence, this example assumes the following conditions.

- An SP has been issued with the Locking Template, and that SP is active (has been issued and personalized)
- The `Locking` table contains a row that represents this unencrypted LBA region.
    - This row's LBA region does not overlap any other row's LBA region.

Step1: This `Locking` table row begins with the following conditions:

- `ReEncryptState` is set to `IDLE`.
- `RangeStart` contains first unencrypted LBA
- `RangeLength` contains the relative offset from first LBA (last LBA to be encrypted).
- `ReadLockEnabled` and `WriteLockEnabled` are configured based upon the associated authority's requirements.
- `ActiveKey` is set to `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00`.

Step2: The following values are configured preceding re-encryption:

- `ContOnReset`: Each entry in this list defines the conditions for continuing the re-encryption process. Ex: if list Null value, then re-encryption is quiesced after a power cycle occurs.
- `VerifyMode` : when value is 1, a read verify operation must be performed on the newly encrypted data after it has been written to the media.
- `AdvKeyMode`: This field defines how the re-encryption exits `ACTIVE` or `COMPLETED` states
- New User_Data encryption key must be loaded into associated `C_*` table.
- `NextKey` is loaded with a pointer to the associated `C_*` table.

Step3: Re-encryption is invoked

- The host sets the value of the `ReEncryptRequest` column to `START_req`.
- When the TPer detects this new value, the `ReEncryptState` value becomes `PENDING`.
- When the TPer, begins re-encryption, the `ReEncryptState` value becomes `ACTIVE`.
- As re-encryption is executing, the `LastReEncryptLBA` is updated. Update rate is implementation specific. However, this value is valid when `ReEncryptState` values are `PENDING`, `PAUSED` or `COMPLETED`.

Step4: Re-encryption is performed. Re-encryption may end in the following way:

- Successful results
    - `ReEncryptState` value changes to `COMPLETED` OR `IDLE` based upon `AdvKeyMode`.
    - `LastReEncryptLBA` equals (`RangeStart + RangeLength - 1`).

  - If COMPLETED state, Host sets ReEncryptRequest to ADVKEY_req. The TPer changes ReEncryptState value to IDLE.

- Failed results
  - ReEncryptState value changes to PAUSED
  - LastReEncryptLBA represents the last good encrypted LBA.
  - PauseStatus represents the condition for changing ReEncryptState to PAUSED.
  - The host defines the recovery steps.

5.8.6.1.2 Encrypt Key 1 to Encrypt Key 2

The following sequence applies to an encrypted User Data LBA region and encryption key must be changed. Preceding this sequence, this example assumes the following conditions.

- An SP has been issued with the Locking Template, and that SP is active (has been issued and personalized)
- The Locking table contains a row that represents this encrypted LBA region.
  - This row's LBA region does not overlap any other row's LBA region.
  - Associated credentials rows in appropriate C_* tables are valid.
- This LBA region contains previously encrypted User_data

Generally, Encrypt Key 1 to Encrypt Key 2 follows steps 1&2 from Section 5.8.6.1.1 except for the following conditions.

- ActiveKey has a pointer to a valid C_* object.

From this point, re-encryption invocation and results are the same as Section 5.8.6.1.1.

5.8.6.1.3 Encrypt Key 1 to Clear Text

Generally, Encrypt to Clear Text User_data steps follow Section 5.8.6.1.2 except for the following conditions.

- The NextKey value is 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00.

When the TPer detects a NextKey column value of zeroes and ReEncryptState changes to START_req, the TPer knows to convert the encrypted User_data LBA region to clear text User_data LBA region.

From this point, re-encryption invocation and results are the same as Section 5.8.6.1.1.

6 Appendix 1 – Required UID Assignments

6.1 Required UID Assignments Overview

The tables in this section define the required UID assignments for objects, methods, and tables, and table rows as required by this specification.

6.2 Reserved UIDs

The first 2^{16} + 1 uids in each table are reserved for use by the TCG. This represents reservation of the lower 4 bytes of each uid, `0x00 0x00 0x00 0x00` to `0x00 0x01 0x00 0x00` inclusive. Reservation allows categorization of table rows using the lower order bytes of each UID.

These default UIDs enable grouping for the following purposes:

- To categorize rows of the `Table` table by Template
- To categorize rows of the `MethodID` table by Template
- To categorize rows of the `Type` table by type format

Each of these categories in each of the respective tables is grouped into 128 categories with 512 UIDs reserved for each category. The following tables identify the values reserved, and the associated category for which they are reserved. Note that the reserved byte values represent the lower 4 bytes of each UID. For additional information on UID assignment, see section 3.2.5.3.

Table 176 MethodID Table and Table Table Reserved LSB Value Ranges

Template Name	Reserved Start	Reserved End
Base	00 00 00 01	00 00 02 00
Admin	00 00 02 01	00 00 04 00
Clock	00 00 04 01	00 00 06 00
Crypto	00 00 06 01	00 00 08 00
Locking	00 00 08 01	00 00 0A 00
Log	00 00 0A 01	00 00 0C 00
Unassigned	00 00 0C 01	00 01 00 00
Unassigned	00 00 00 00	

Table 177 Type Table Reserved LSB Value Ranges

Type Format Name	Reserved Start	Reserved End
Base Type	00 00 00 01	00 00 02 00
Simple Type	00 00 02 01	00 00 04 00
Enumeration Type	00 00 04 01	00 00 06 00
Alternative Type	00 00 06 01	00 00 08 00
List Type	00 00 08 01	00 00 0A 00
Restricted Reference Type – Row Ref	00 00 0A 01	00 00 0C 00
Restricted Reference Type – UID	00 00 0C 01	00 00 0F 00
General Reference Type – Row Ref	00 00 0F 01	00 00 10 00
General Reference Type – Object UID	00 00 10 01	00 00 12 00
General Reference Type – Table UID	00 00 12 01	00 00 14 00
Name Value Type	00 00 14 01	00 00 16 00
Struct Type	00 00 16 01	00 00 18 00

Type Format Name	Reserved Start	Reserved End
Set Type	00 00 18 01	00 00 1A 00
Unassigned	00 00 1A 01	00 01 00 00
Unassigned	00 00 00 00	

6.3 Assigned UIDs

The tables in this section display the assigned UIDs required to be used with the associated tables, methods, objects, and table rows. For additional information on UID assignment, see section 3.2.5.3.

The tables in this section describe:

- Special Purpose UIDs (Table 178) – this descriptive table contains UIDs assigned special meanings/functions in the Core Specification, and a brief description of their functions.
- Table UIDs (Table 179) – this descriptive table contains UIDs assigned to all table descriptor objects (objects in the `Table` table), as well as the UIDs assigned to the tables themselves.
- Session Manager Method UIDs (Table 180) – this descriptive table contains the UIDs assigned to Session Manager layer methods.
- MethodID UIDs (Table 181) – this descriptive table contains the UIDs assigned in the `MethodID` table to all preinstalled methods.
- Authority UIDs (Table 182) – this descriptive table contains the UIDs assigned in the `Authority` table to each of the default authorities described in the Core Spec.
- ACE UIDs (Table 183) – this descriptive table contains the UIDs assigned in the `ACE` table to each of the default ACEs described in the Life Cycle/Default Access Control sections of the TCG Core Specification.
- Single Row Table UIDs (Table 184) – this descriptive table contains the UIDs assigned to rows in the tables described in the TCG Core Specification as having only one row.
- Table Default Rows (Table 185) – In some instances, the TCG Core Specification also defines the UIDs of certain objects within some tables. This descriptive table contains the UIDs assigned to these objects.
- Type UIDs (Table 186) – this descriptive table contains the UIDs assigned to all of the predefined types, as identified in the `Type` table. The descriptive table also identifies the table its columns associated with each type.
- Template Table UIDs (Table 187) – this descriptive table contains the UIDs assigned to all of the Templates defined in this specification that would appear in the Admin SP's `Template` table.
- SPTemplates Table UIDs (Table 188) – this descriptive table contains the UIDs assigned to all of the Templates defined in this specification that would appear in an SP's `SPTemplates` table.

Table 178 Special Purpose UIDs

UID	Purpose
00 00 00 00 00 00 00 00	Used to represent Null UID
00 00 00 00 00 00 00 01	Used as the SPUID, the UID that identifies "This SP" – used as the InvokingID for invocation of SP methods
00 00 00 00 00 00 00 FF	Used as the SMUID, the UID that identifies "the Session manager" – used as InvokingID for invocation of Session Manager layer methods

UID	Purpose
00 00 00 00 00 00 FF xx	Identifies UIDs assigned to Session Manager layer methods, where xx is the UID assigned to a particular method (see Table 180)

Table 179 Table UIDs

UID of Table Descriptor Object	UID of Table	Table Name	Template
00 00 00 01 00 00 00 01	00 00 00 01 00 00 00 00	Table	Base
00 00 00 01 00 00 00 02	00 00 00 02 00 00 00 00	SPInfo	Base
00 00 00 01 00 00 00 03	00 00 00 03 00 00 00 00	SPTemplates	Base
00 00 00 01 00 00 00 04	00 00 00 04 00 00 00 00	Column	Base
00 00 00 01 00 00 00 05	00 00 00 05 00 00 00 00	Type	Base
00 00 00 01 00 00 00 06	00 00 00 06 00 00 00 00	MethodID	Base
00 00 00 01 00 00 00 07	00 00 00 07 00 00 00 00	Method	Base
00 00 00 01 00 00 00 08	00 00 00 08 00 00 00 00	ACE	Base
00 00 00 01 00 00 00 09	00 00 00 09 00 00 00 00	Authority	Base
00 00 00 01 00 00 00 0A	00 00 00 0A 00 00 00 00	Certificates	Base
00 00 00 01 00 00 00 0B	00 00 00 0B 00 00 00 00	C_PIN	Base
00 00 00 01 00 00 00 0C	00 00 00 0C 00 00 00 00	C_RSA_1024	Base
00 00 00 01 00 00 00 0D	00 00 00 0D 00 00 00 00	C_RSA_2048	Base
00 00 00 01 00 00 00 0E	00 00 00 0E 00 00 00 00	C_AES_128	Base
00 00 00 01 00 00 00 0F	00 00 00 0F 00 00 00 00	C_AES_256	Base
00 00 00 01 00 00 00 10	00 00 00 10 00 00 00 00	C_EC_160	Base
00 00 00 01 00 00 00 11	00 00 00 11 00 00 00 00	C_EC_192	Base
00 00 00 01 00 00 00 12	00 00 00 12 00 00 00 00	C_EC_224	Base
00 00 00 01 00 00 00 13	00 00 00 13 00 00 00 00	C_EC_256	Base
00 00 00 01 00 00 00 14	00 00 00 14 00 00 00 00	C_EC_384	Base
00 00 00 01 00 00 00 15	00 00 00 15 00 00 00 00	C_EC_521	Base
00 00 00 01 00 00 00 16	00 00 00 16 00 00 00 00	C_EC_163	Base
00 00 00 01 00 00 00 17	00 00 00 17 00 00 00 00	C_EC_233	Base
00 00 00 01 00 00 00 18	00 00 00 18 00 00 00 00	C_EC_283	Base
00 00 00 01 00 00 00 19	00 00 00 19 00 00 00 00	C_HMAC_160	Base
00 00 00 01 00 00 00 1A	00 00 00 1A 00 00 00 00	C_HMAC_256	Base
00 00 00 01 00 00 00 1B	00 00 00 1B 00 00 00 00	C_HMAC_384	Base
00 00 00 01 00 00 00 1C	00 00 00 1C 00 00 00 00	C_HMAC_512	Base
00 00 00 01 00 00 02 01	00 00 02 01 00 00 00 00	TPerInfo	Admin
00 00 00 01 00 00 02 02	00 00 02 02 00 00 00 00	Properties	Admin
00 00 00 01 00 00 02 03	00 00 02 03 00 00 00 00	CryptoSuite	Admin
00 00 00 01 00 00 02 04	00 00 02 04 00 00 00 00	Template	Admin
00 00 00 01 00 00 02 05	00 00 02 05 00 00 00 00	SP	Admin
00 00 00 01 00 00 04 01	00 00 04 01 00 00 00 00	ClockTime	Clock
00 00 00 01 00 00 06 01	00 00 06 01 00 00 00 00	H_SHA_1	Crypto
00 00 00 01 00 00 06 02	00 00 06 02 00 00 00 00	H_SHA_256	Crypto
00 00 00 01 00 00 06 03	00 00 06 03 00 00 00 00	H_SHA_384	Crypto

UID of Table Descriptor Object	UID of Table	Table Name	Template
00 00 00 01 00 00 06 04	00 00 06 04 00 00 00 00	H_SHA_512	Crypto
00 00 00 01 00 00 0A 01	00 00 0A 01 00 00 00 00	Log	Log
00 00 00 01 00 00 0A 02	00 00 0A 02 00 00 00 00	LogList	Log
00 00 00 01 00 00 08 01	00 00 08 01 00 00 00 00	LockingInfo	Locking
00 00 00 01 00 00 08 02	00 00 08 02 00 00 00 00	Locking	Locking
00 00 00 01 00 00 08 03	00 00 08 03 00 00 00 00	MBRControl	Locking
00 00 00 01 00 00 08 04	00 00 08 04 00 00 00 00	MBR	Locking

Table 180 Session Manager Method UIDs

Method UID	Method Name
00 00 00 00 00 00 FF 01	Properties
00 00 00 00 00 00 FF 02	StartSession
00 00 00 00 00 00 FF 03	SyncSession
00 00 00 00 00 00 FF 04	StartTrustedSession
00 00 00 00 00 00 FF 05	SyncTrustedSession
00 00 00 00 00 00 FF 06	CloseSession

Table 181 MethodID UIDs

UID in MethodID Table	Method Name	Template
00 00 00 06 00 00 00 01	DeleteSP	Base
00 00 00 06 00 00 00 02	CreateTable	Base
00 00 00 06 00 00 00 03	Delete	Base
00 00 00 06 00 00 00 04	CreateRow	Base
00 00 00 06 00 00 00 05	DeleteRow	Base
00 00 00 06 00 00 00 06	Get	Base
00 00 00 06 00 00 00 07	Set	Base
00 00 00 06 00 00 00 08	Next	Base
00 00 00 06 00 00 00 09	GetFreeSpace	Base
00 00 00 06 00 00 00 0A	GetFreeRows	Base
00 00 00 06 00 00 00 0B	DeleteMethod	Base
00 00 00 06 00 00 00 0C	Authenticate	Base
00 00 00 06 00 00 00 0D	GetACL	Base
00 00 00 06 00 00 00 0E	AddACE	Base
00 00 00 06 00 00 00 0F	RemoveACE	Base
00 00 00 06 00 00 00 10	GenKey	Base
00 00 00 06 00 00 02 01	IssueSP	Admin
00 00 00 06 00 00 04 01	GetClock	Clock
00 00 00 06 00 00 04 02	ResetClock	Clock
00 00 00 06 00 00 04 03	SetClockHigh	Clock
00 00 00 06 00 00 04 04	SetLagHigh	Clock
00 00 00 06 00 00 04 05	SetClockLow	Clock

UID in MethodID Table	Method Name	Template
00 00 00 06 00 00 04 06	SetLagLow	Clock
00 00 00 06 00 00 04 07	IncrementCounter	Clock
00 00 00 06 00 00 06 01	Random	Crypto
00 00 00 06 00 00 06 02	Salt	Crypto
00 00 00 06 00 00 06 03	DecryptInit	Crypto
00 00 00 06 00 00 06 04	Decrypt	Crypto
00 00 00 06 00 00 06 05	DecryptFinalize	Crypto
00 00 00 06 00 00 06 06	EncryptInit	Crypto
00 00 00 06 00 00 06 07	Encrypt	Crypto
00 00 00 06 00 00 06 08	EncryptFinalize	Crypto
00 00 00 06 00 00 06 09	HMACInit	Crypto
00 00 00 06 00 00 06 0A	HMACCalc	Crypto
00 00 00 06 00 00 06 0B	HMACFinalize	Crypto
00 00 00 06 00 00 06 0C	HashInit	Crypto
00 00 00 06 00 00 06 0D	HashCalc	Crypto
00 00 00 06 00 00 06 0E	HashFinalize	Crypto
00 00 00 06 00 00 06 0F	Sign	Crypto
00 00 00 06 00 00 06 10	Verify	Crypto
00 00 00 06 00 00 06 11	XOR	Crypto
00 00 00 06 00 00 0A 01	AddLog	Log
00 00 00 06 00 00 0A 02	CreateLog	Log
00 00 00 06 00 00 0A 03	ClearLog	Log
00 00 00 06 00 00 0A 04	FlushLog	Log
00 00 00 06 00 00 08 01	GetPackage	Locking
00 00 00 06 00 00 08 02	SetPackage	Locking

Table 182 Authority UIDs

UID in Authority Table	Authority Name	Template
00 00 00 09 00 00 00 01	Anybody	Base
00 00 00 09 00 00 00 02	Admins	Base
00 00 00 09 00 00 00 03	Makers	Base
00 00 00 09 00 00 00 04	MakerSymK	Base
00 00 00 09 00 00 00 05	MakerPuK	Base
00 00 00 09 00 00 00 06	SID	Base
00 00 00 09 00 00 00 07	TPerSign	Base
00 00 00 09 00 00 00 08	TPerExch	Base
00 00 00 09 00 00 00 09	AdminExch	Base
00 00 00 09 00 00 02 01	Issuers	Admin
00 00 00 09 00 00 02 02	Editors	Admin
00 00 00 09 00 00 02 03	Deleters	Admin
00 00 00 09 00 00 02 04	Servers	Admin
00 00 00 09 00 00 02 05	Reserve0	Admin
00 00 00 09 00 00 02 06	Reserve1	Admin
00 00 00 09 00 00 02 07	Reserve2	Admin

00 00 00 09 00 00 02 08	Reserve3	Admin

Table 183 ACE UIDs

UID in ACE Table	ACE Name	Template
00 00 00 08 00 00 00 01	Anybody	Base
00 00 00 08 00 00 00 02	Admins	Base
00 00 00 08 00 00 00 03	Makers	Base
00 00 00 08 00 00 00 04	PostIssuanceAdmins	Base
00 00 00 08 00 00 00 05	SPInfo_1	Base
00 00 00 08 00 00 00 06	Table_Size	Base
00 00 00 08 00 00 00 07	Method_1	Base
00 00 00 08 00 00 00 08	Method_2	Base
00 00 00 08 00 00 02 01	SID	Admin
00 00 00 08 00 00 02 02	Issuers	Admin
00 00 00 08 00 00 02 03	Editors	Admin
00 00 00 08 00 00 02 04	Deleters	Admin
00 00 00 08 00 00 02 05	Servers	Admin
00 00 00 08 00 00 02 06	Issuers_SID	Admin
00 00 00 08 00 00 0A 01	LogList_Security	Log
00 00 00 08 00 00 08 01	LockingInfo_1	Locking
00 00 00 08 00 00 08 02	Locking_1	Locking
00 00 00 08 00 00 08 03	Locking_2	Locking
00 00 00 08 00 00 08 04	Locking_3	Locking

Table 184 Single Row Table Row UIDs

UID of Row	Single Row Table Name
00 00 00 02 00 00 00 01	SPInfo
00 00 02 01 00 00 00 01	TPerInfo
00 00 08 01 00 00 00 01	LockingInfo
00 00 08 03 00 00 00 01	MBR_Control

Table 185 Table Default Rows

UID of Row	Table Name	Row Name
00 00 00 0B 00 00 00 01	C_PIN	SID
00 00 02 05 00 00 00 01	SP	Admin
00 00 04 01 00 00 00 01	ClockTime	Clock
00 00 0A 02 00 00 00 01	LogList	Log
00 00 08 02 00 00 00 01	Locking	Global Range

Table 186 Type UIDs

Type UID	Type Name	Table.Column
00 00 00 05 00 00 00 01	NULL	
00 00 00 05 00 00 00 02	bytes	
00 00 00 05 00 00 00 03	max_bytes	
00 00 00 05 00 00 00 04	integer	
00 00 00 05 00 00 00 05	uinteger	
00 00 00 05 00 00 02 01	bytes_12	TPerInfo.GUDID
00 00 00 05 00 00 02 02	bytes_16	C_AES_128.Key, C_AES_128.ResidualData
00 00 00 05 00 00 02 03	bytes_20_def_00	H_SHA_1.Proof, H_SHA_1.Accumulator
00 00 00 05 00 00 02 04	bytes_32_def_00	H_SHA_256.Proof, H_SHA_1.Accumulator
00 00 00 05 00 00 02 05	bytes_32	C_AES_256.Key, C_AES_256.ResidualData, SP.EffectiveAuth
00 00 00 05 00 00 02 06	version_bytes_4	SPTemplates.Version
00 00 00 05 00 00 02 07	bytes_48_def_00	H_SHA_384.Proof, H_SHA_384.Accumulator
00 00 00 05 00 00 02 08	bytes_64_def_00	H_SHA_512.Proof, H_SHA_512.Accumulator
00 00 00 05 00 00 02 09	uid	All.UID, SPInfo.SPID, SPTemplates.TemplateID, Table.LastID
00 00 00 05 00 00 02 0A	certificate	
00 00 00 05 00 00 02 0B	name	All.name, All.CommonName, ACE.ColStart, ACE.ColEnd, CryptoSuite.CryptoCall, CryptoSuite.CryptoOp, Log.SigningAuthName, Log.ExchangeAuthName
00 00 00 05 00 00 02 0C	password	C_PIN.PIN
00 00 00 05 00 00 02 0D	max_bytes_32	Locking.LastReEncryptLBA
00 00 00 05 00 00 02 0E	max_bytes_64	
00 00 00 05 00 00 02 0F	int_1_def_0	C_RSA_*.ChainLimit
00 00 00 05 00 00 02 10	integer_1	
00 00 00 05 00 00 02 11	uinteger_1	
00 00 00 05 00 00 02 12	uinteger_128	C_RSA_1024.Pu_Exp/Mod/Pr_exp, C_RSA_2048.P/Q/Dmp1/Dmq1/Iqmp
00 00 00 05 00 00 02 13	uinteger_16	C_AES_128.Key
00 00 00 05 00 00 02 14	feedback_size	C_AES_*.FeedbackSize
00 00 00 05 00 00 02 15	uinteger_2	Type.Size, CryptoSuite.CryptoLen, Template.Instances, Template.MaxInstances
00 00 00 05 00 00 02 16	uinteger_20	
00 00 00 05 00 00 02 17	uinteger_21	

Type UID	Type Name	Table.Column
00 00 00 05 00 00 02 18	uinteger_24	
00 00 00 05 00 00 02 19	uinteger_256	C_RSA_2048.Pu_Exp/Mod/Pr_exp
00 00 00 05 00 00 02 1A	uinteger_28	
00 00 00 05 00 00 02 1B	uinteger_30	
00 00 00 05 00 00 02 1C	challenge	
00 00 00 05 00 00 02 1D	uinteger_32	
00 00 00 02 00 00 02 1E	max_bytes_get	
00 00 00 05 00 00 02 1F	uinteger_36	
00 00 00 05 00 00 02 20	uinteger_4	All Array.RowNumber, SPInfo.SPSessionTimeout, Table.NumColumns, Table.Rows, Table.RowsFree, Table.RowBytes, Table.MinSize, Column.Byte, CryptoSuite.Time, CryptoSuite.Variance, Template.RevisionNumber, Log.Session, Locking_Info.MaxRanges, Locking_Info.MaxReEncryptions, Certificates.CertSize, TPerInfo.Generation, TPerInfo.FirmwareVersion, TPerInfo.ProtocolVersion, Locking_Info.Version, C_PIN.TryLimit, C_PIN.Tries
00 00 00 05 00 00 02 21	uint_4_def_0	Table.MaxSize, Authority.Limit, Authority.Uses
00 00 00 05 00 00 02 22	max_bytes_set	
00 00 00 05 00 00 02 23	uinteger_48	
00 00 00 05 00 00 02 24	uinteger_64	C_RSA_1024.P/Q/Dmp1/Dmq1/Iqmp
00 00 00 05 00 00 02 25	uinteger_8	SPInfo.Size, SPInfo.SizeInUse, TPerInfo.Bytes, TPer.SpaceForIssuance, SP.Bytes, ClockTime.MonotonicBase, ClockTime.MonotonicReserve, Log.MonotonicTime, Locking.RangeStart, Locking.RangeLength
00 00 00 05 00 00 02 26	common_name	All.CommonName
00 00 00 05 00 00 02 27	uinteger_66	C_EC_521
00 00 00 05 00 00 02 28	signed_hash	
00 00 00 05 00 00 02 29	response	
00 00 00 05 00 00 02 2A	session_key_encrypt	
00 00 00 05 00 00 02 2B	session_key_integrity	
00 00 00 05 00 00 02 2C	proof	
00 00 00 05 00 00 02 2D	exchange_key	
00 00 00 05 00 00 02 2E	iv	
00 00 00 05 00 00 02 2F	encrypt_result	

Type UID	Type Name	Table.Column
00 00 00 05 00 00 02 30	decrypt_result	
00 00 00 05 00 00 02 31	sign_result	
00 00 00 05 00 00 02 32	hash_result	
00 00 00 05 00 00 02 33	hmac_result	
00 00 00 05 00 00 02 34	xor_result	
00 00 00 05 00 00 02 35	max_bytes_256	
00 00 00 05 00 00 02 36	bytes_20	
00 00 00 05 00 00 02 37	bytes_48	
00 00 00 05 00 00 02 38	bytes_64	
00 00 00 05 00 00 02 39	encrypt_max_bytes_input	
00 00 00 05 00 00 02 3A	decrypt_max_bytes_input	
00 00 00 05 00 00 02 3B	sign_max_bytes_input	
00 00 00 05 00 00 02 3C	verify_max_bytes_input	
00 00 00 05 00 00 02 3D	verify_max_bytes_proof	
00 00 00 05 00 00 02 3E	hash_max_bytes_input	
00 00 00 05 00 00 02 3F	hmac_max_bytes_input	
00 00 00 05 00 00 02 40	xor_max_bytes_input	
00 00 00 05 00 00 02 41	stir_integer	
00 00 00 05 00 00 04 01	boolean	SPInfo.Enabled, Authority.IsClass, Authority.PresentCertificate, ClockTime.HaveHigh, ClockTime.HaveLow, LogList.HighSecurity, Locking.ReadLocked, Locking.WriteLocked
00 00 00 05 00 00 04 02	boolean_def_false	Column.IsIndex, CryptoSuite.Special, SP.Frozen, Locking.ReadLockEnabled, Locking.WriteLockEnabled
00 00 00 05 00 00 04 03	boolean_def_true	Column.Transactional, Authority.Enabled
00 00 00 05 00 00 04 04	messaging_type	Authority.Secure
00 00 00 05 00 00 04 05	life_cycle_state	SP.LifeCycleState
00 00 00 05 00 00 04 06	padding_type	C_RSA_*.Format
00 00 00 05 00 00 04 08	auth_method	Authority.Operation
00 00 00 05 00 00 04 09	log_kind	Log.LogKind
00 00 00 05 00 00 04 0A	symmetric_mode	C_AES_*.Mode
00 00 00 05 00 00 04 0B	clock_kind	ClockTime.TrustMode, Log.TimeKind
00 00 00 05 00 00 04 0C	log_select	Method.Log, Method.AddACELog, Method.RemoveACELog, Method.GetACLLog, Method.DeleteMethodLog, Authority.Log
00 00 00 05 00 00 04 0D	hash_protocol	Authority.HashAndSign, C_*.Hash

Type UID	Type Name	Table.Column
00 00 00 05 00 00 04 0E	boolean_ACE	
00 00 00 05 00 00 04 0F	adv_key_mode	Locking.AdvKeyMode
00 00 00 05 00 00 04 10	keys_avail_conds	Locking_Info.KeysAvailCfg
00 00 00 05 00 00 04 11	last_reenc_stat	Locking.LastReEncStat
00 00 00 05 00 00 04 12	verify_mode	Locking.VerifyMode
00 00 00 05 00 00 04 13	reencrypt_request	Locking.ReEncryptRequest
00 00 00 05 00 00 04 14	reencrypt_state	Locking.ReEncryptState
00 00 00 05 00 00 04 15	table_kind	Table.Kind
00 00 00 05 00 00 04 16	package_purpose	
00 00 00 05 00 00 06 01	ACE_expression	
00 00 00 05 00 00 06 02	row_selection	ACE.RowStart, ACE.RowEnd
00 00 00 05 00 00 06 03	columns	
00 00 00 05 00 00 06 04	uint_ref	
00 00 00 05 00 00 06 05	row	
00 00 00 05 00 00 06 06	table_object_ref	Method.InvokingID
00 00 00 05 00 00 06 07	createrow_result	
00 00 00 05 00 00 06 08	next_result	
00 00 00 05 00 00 06 09	get_result	
00 00 00 05 00 00 06 0A	set_values	
00 00 00 05 00 00 06 0B	encrypt_input	
00 00 00 05 00 00 06 0C	decrypt_input	
00 00 00 05 00 00 06 0D	sign_input	
00 00 00 05 00 00 06 0E	verify_input	
00 00 00 05 00 00 06 0F	verify_proof	
00 00 00 05 00 00 06 10	hash_input	
00 00 00 05 00 00 06 11	hmac_input	
00 00 00 05 00 00 06 12	xor_input	
00 00 00 05 00 00 06 13	stir_input	
00 00 00 05 00 00 06 14	challenge	
00 00 00 05 00 00 08 01	AC_element	ACE.BooleanExpr
00 00 00 05 00 00 08 02	ACL	Method.ACL, Method.AddACEACL, Method.RemoveACEACL, Method.GetACLACL, Method.DeleteMethodACL
00 00 00 05 00 00 08 03	type_ref_list	
00 00 00 05 00 00 08 04	row_data	
00 00 00 05 00 00 08 05	columns_list	
00 00 00 05 00 00 08 06	uint_ref_list	
00 00 00 05 00 00 08 07	template_list	
00 00 00 05 00 00 08 08	ref_uidref_createrow_list	
00 00 00 05 00 00 08 09	uidref_createrow_list	
00 00 00 05 00 00 08 0A	ref_uidref_next_list	
00 00 00 05 00 00 08 0B	uidref_next_list	
00 00 00 05 00 00 08 0C	Table_ref_rows_list	
00 00 00 05 00 00 08 0D	get_column_sub_list	

Type UID	Type Name	Table.Column
00 00 00 05 00 00 08 0E	get_column_list	
00 00 00 05 00 00 08 0F	set_column_sub_list	
00 00 00 05 00 00 08 10	set_column_list	
00 00 00 05 00 00 0A 01	column_ref	Column.Next, Table.Column
00 00 00 05 00 00 0C 01	SPTemplates_ref	All.TemplateID
00 00 00 05 00 00 0C 02	Type_ref	Column.Type
00 00 00 05 00 00 0C 03	MethodID_ref	MethodID.MethodID
00 00 00 05 00 00 0C 04	ACE_table_ref	
00 00 00 05 00 00 0C 05	Authority_ref	Authority.Class, Authority.ResponseSign, Authority.ResponseExch, SP.ORG, ClockTime.HaveByWhom, ClockTime.LowByWhom, Log.SigningAuthority, Log.ExchangeAuthority
00 00 00 05 00 00 0C 06	Certificates_ref	C_RSA_*.Certificate, C_EC_*.Certificate
00 00 00 05 00 00 0C 07	SP_ref	
00 00 00 05 00 00 0C 08	Template_ref	SPTemplates.TemplateID
00 00 00 05 00 00 0C 09	Table_ref	
00 00 00 05 00 00 0F 01	row_ref	
00 00 00 05 00 00 0F 02	log_row_ref	LogList.Serial
00 00 00 05 00 00 10 01	row_uidref	
00 00 00 05 00 00 10 02	cred_object_uidref	Authority.Credential, H_SHA_*.Signer, Locking.ActiveKey, Locking.NextKey
00 00 00 05 00 00 12 01	table_ref	Certificates.CertData, LogList.Log
00 00 00 05 00 00 12 02	ref_def_00	Type.Default, Method.LogTo, Authority.LogTo, C_PIN.CharSet
00 00 00 05 00 00 12 03	byte_table_ref	Certificates.CertData
00 00 00 05 00 00 14 01	column-name	
00 00 00 05 00 00 14 02	row_uidref-name	
00 00 00 05 00 00 14 03	row_ref-name	
00 00 00 05 00 00 14 04	table_ref-name	
00 00 00 05 00 00 14 05	type_ref-name	
00 00 00 05 00 00 14 06	name-uinteger_2	
00 00 00 05 00 00 14 07	name-uinteger_1	
00 00 00 05 00 00 14 08	name-startColumn	
00 00 00 05 00 00 14 09	name-endColumn	
00 00 00 05 00 00 14 0A	name-startRow	
00 00 00 05 00 00 14 0B	name-endRow	
00 00 00 05 00 00 16 01	column	
00 00 00 05 00 00 16 02	lag	ClockTime.HighLag, ClockTime.LowLag

Type UID	Type Name	Table.Column
00 00 00 05 00 00 16 03	columns_struct	
00 00 00 05 00 00 16 04	date	Authority.ClockStart, Authority.ClockEnd, SP.DateofIssue
00 00 00 05 00 00 16 05	clock_time	ClockTime.HighSetTime, ClockTime.HighInitialTimer; ClockTime.LowSetTime, ClockTime.LowInitialTimer, Log.ExactTime
00 00 00 05 00 00 16 06	type_def	Type.Format
00 00 00 05 00 00 16 07	cell_block	
00 00 00 05 00 00 16 08	struct-name-uinteger_1	
00 00 00 05 00 00 16 09	struct-ref_uidref	
00 00 00 05 00 00 16 0A	struct-Table_ref_uint_4	
00 00 00 05 00 00 18 01	reset_types	Locking.LockOnReset, Locking.ContOnReset
00 00 00 05 00 00 18 02	gen_status	Locking.GeneralStatus
00 00 00 05 00 00 18 03	enc_supported	Locking_Info.EncryptSupport

Table 187 Template Table UIDs

UID of Row	Template Name
00 00 02 04 00 00 00 01	Base
00 00 02 04 00 00 00 02	Admin
00 00 02 04 00 00 00 03	Clock
00 00 02 04 00 00 00 04	Crypto
00 00 02 04 00 00 00 05	Log
00 00 02 04 00 00 00 06	Locking

Table 188 SPTemplates Table UIDs

UID of Row	SPTemplates Name
00 00 00 03 00 00 00 01	Base
00 00 00 03 00 00 00 02	Admin
00 00 00 03 00 00 00 03	Clock
00 00 00 03 00 00 00 04	Crypto
00 00 00 03 00 00 00 05	Log
00 00 00 03 00 00 00 06	Locking

[C D1]TC: known/expected behaviour (remove?)
[ms2]This only works if the previous packet i.e. N-1 has the same payload, is this what is meant here?
[ms3]Should this be “Packet N+1”