TCG Infrastructure Working Group Platform Trust Services Interface Specification (IF-PTS)

Specification Version 1.0
Revision 1.0
17 November 2006
FINAL

Contact:

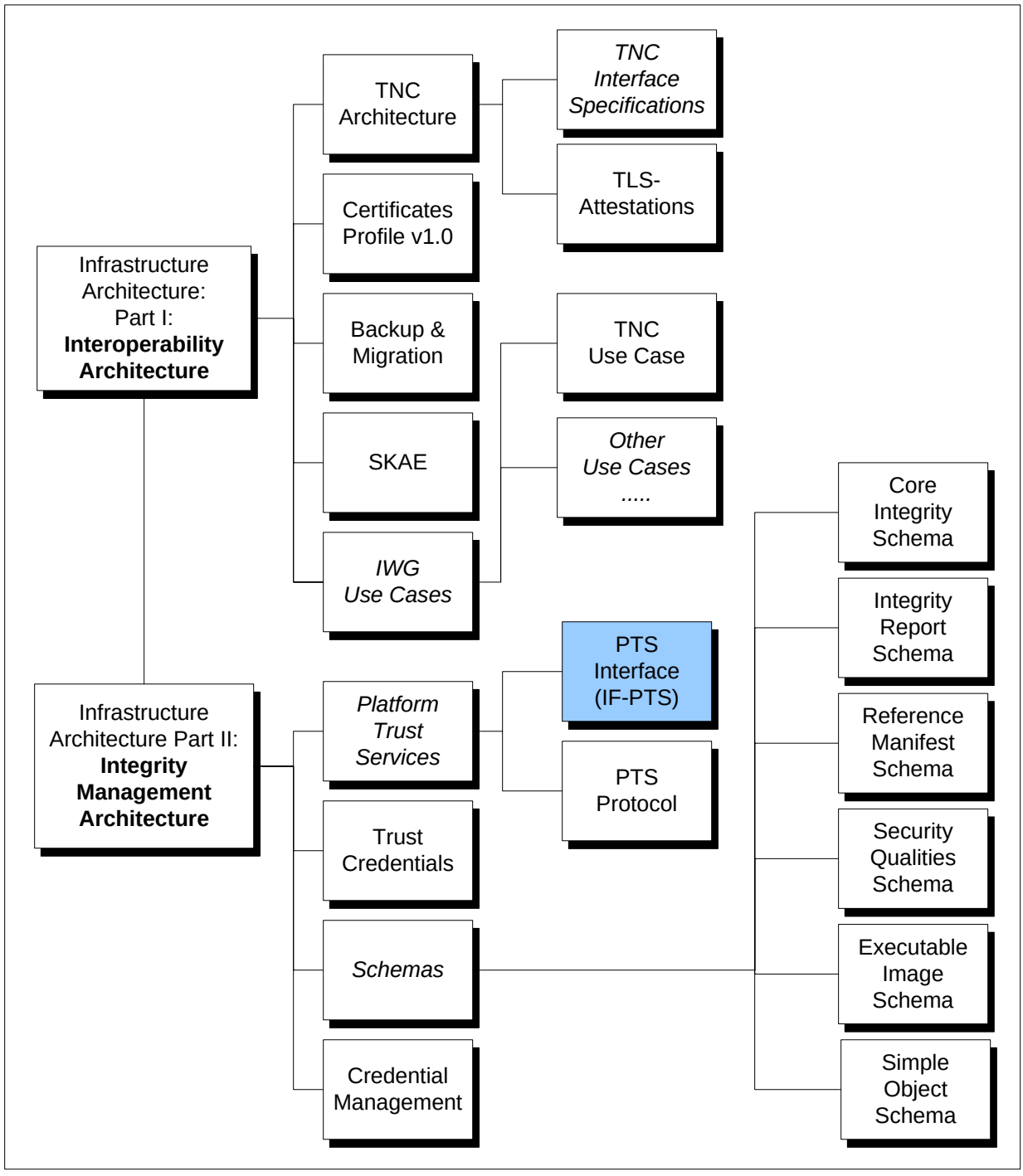
ned.smith@intel.com (Editor, IWG Co-Chair)
GregK@wavesys.com (Editor)
THardjono@SignaCert.com (IWG Co-Chair)
Shanna@juniper.net (TNC Co-Chair)
Paul_Sangster@symantec.com (TNC Co-Chair)

TCG

Public

Disclaimer

IWG TNC Document Roadmap



Acknowledgement

The TCG wishes to thank all those who contributed to this specification. This document builds on considerable work done in the various working groups in the TCG.

Special thanks to the members of the IWG contributing to this document:

Name	Company
Malcolm Duncan	**CESG**
Diana Arroyo	**IBM**
Lee Terrell	**IBM**
Markus Gueller	**Infineon**
Ned Smith (Editor, Co-Chair)	**Intel Corporation**
Mark Williams	**Microsoft**
Thomas Hardjono (Co-Chair)	**Signacert**
Jeff Nisewanger	**Sun**
Wyllys Ingersoll	**Sun**
Paul Sangster (TNC Co-Chair)	**Symantec**
Greg Kazmierczak (Editor)	**Wave Systems**
Len Veil	**Wave Systems**

Table of Contents

Table of Figures

1 Scope and Audience

The Infrastructure Working Group (IWG) defines architecture and frameworks for interoperation of the trusted computing capabilities of platforms and devices useful in improving the security and assurance of computer systems. Endpoint integrity is critical to network connectivity where an access control decision is made. The TCG defines architecture and specifications that enable network operators to enforce endpoint integrity when granting access to a network infrastructure. The TNC architecture incorporates the IF-PTS interface to leverage Platform Trust Services (PTS) using TNC Integrity Measurement Collectors (IMC) and other system components that may be involved whenever an evaluation of endpoint integrity factors into access control decisions.

This specification defines the IF-PTS interface at the protocol data unit (PDU) level..Interoperability is achieved through compliance with PDU structure definition and proper message passing semantics.

Architects, designers, developers and technologists who wish to implement, use, or understand IF-PTS should read this document carefully. Before reading this document, the reader should review and understand the TNC architecture specification as described in [1], the IWG Architecture Part II specification and the IWG Integrity Schema family of specification.

1.1 Keywords

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [2].

2 Background

2.1 Purpose of IF-PTS

This document describes and specifies IF-PTS, a critical interface in the Trusted Computing Group's Trusted Network Connect (TNC) architecture and part of any other application framework where establishment of endpoint integrity is needed. IF-PTS can be used by Integrity Measurement Collectors (IMCs), TNC Client (TNCC) and Network Access Requester (NAR) and other clients to report on endpoint integrity state.

IF-PTS can be used in several ways to improve trusted computing goals.

- PTS enables platform components to participate in Platform Transitive Trust chains.
- Computation and collection of integrity measurements over TNC and other application components.
- Formatting of integrity measurements collected by TNC and other applications for interoperability.
- Client side (local) verification of measurements.

PTS collects the integrity status of TNC components such that unauthorized modification to component images can be detected. PTS constructs integrity reports and makes them available to Integrity Measurement Collectors and Verifiers. The integrity state of the TNC framework is combined with the platform's pre-existing transitive-trust measurements such that a chain of trust dependency can be determined and evaluated. Knowledge of TNC subsystem integrity state can be an important precondition to trusting values reported by individual IMCs. Additional detail of the features provided by the IF-PTS API is provided in section 2.5.

PTS produces an integrity report data structure[1] suitable for use by a verification process running on a remote system from the same or another vendor.

PTS may be used to evaluate measurements. By applying rules, locally evaluated measurements can be an effective way to minimize communicating voluminous measurement data over a network or low bandwidth channel. The result of a PTS applied rule is a new measurement value that could be reported to a remote verifier (such as an IMV).

PTS facilitates decision making by the TNC system to both improve assurances of a properly operating TNC and to lower the threshold for interoperation among IMCs and IMVs by constructing integrity reports using a standardized format.

2.2 Architecture

PTS plays a primary role in the client-side architecture providing access to pre-computed integrity values, integrity values computed directly by PTS and formatting of integrity values computed (or provided by) an IMC. PTS may verify integrity values and produce in response, new integrity values that capture the result of having applied a verification policy.

Additional background related to the Integrity Management and PTS architecture is found in the *IWG Architecture for Interoperability Part II v1.0*.

PTS may play a supporting role in the server-side architecture by providing parsing and verification capabilities as suggested by the dotted box by the IMV in Figure 1. This figure depicts an end-to-end architecture where collection and verification originate and terminate respectively with a Platform Trust Service. The Trusted Network Connect (TNC) infrastructure is used to communicate integrity measurements from one platform to the other. The IMC / IMV pair defines an application specific protocol for exchanging integrity measurements.

[1] TCG Integrity Report Schema Specification v1.0

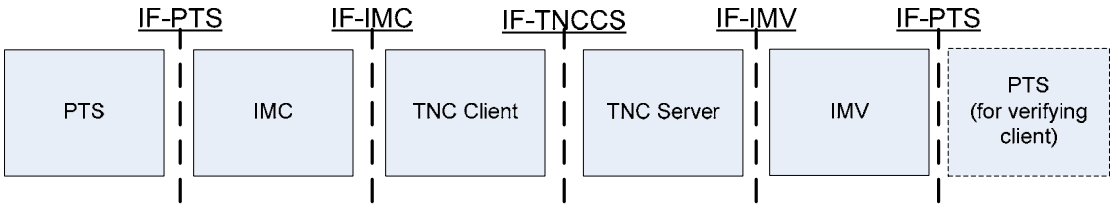


Figure 1 - PTS End-to-End Architecture

The system architecture for the Platform Trust Service (*PTS*) is depicted in Figure 2. The PTS collects measurements from the *TNC-Client plus IMCs*, Network Access Requestor (*NAR*) and possibly other *Processes* and platform components. PTS Measurements may be stored in an *Integrity Measurement Log* where they may be retrieved for later use. The Measurement Log also contains the platform's *Transitive-trust Chain* created by the Root-of-Trust for Measurement (RTM) and other measurement agents that may have executed prior to PTS execution (See Figure 10). Transitive trust is a system for loading code into memory in a way that allows multiple code modules to have cryptographically integrity protected linkages between the loaded code and security components of the platform. Ideally part of the Trusted OS should compute a measurement of the PTS and append it to the *Transitive-trust* chain prior to passing execution to the PTS.

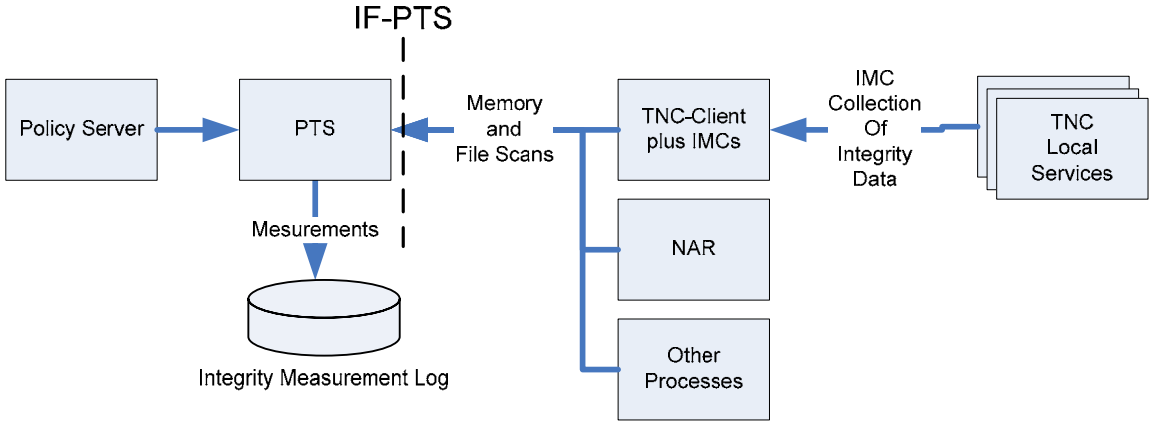


Figure 2 - PTS as a Measurement Agent

The PTS-IMC is a TNC Integrity Measurement Collector (IMC) that interfaces with the PTS to obtain and report the *Transitive-trust Chain* measurements to a PTS-IMV (not shown). The PTS-IMC and PTS-IMV define a suitable protocol for exchanging and ensuring the integrity of the measurements collected by the client. It is expected that PTS-IMC, as well as other processes in system, will use IF-PTS interfaces to interact with the PTS.

Reported measurements are evaluated by a verifier such as a PTS-IMV (Figure 1). A PTS *Policy Server* provides rules describing acceptable and/or unacceptable transitive-trust chain configurations via the IF-PTS interface. Policies provisioned into PTS should be authenticated to an authorized Policy Server..

PTS can support intermediate verification operations by applying an integrity policy to collected integrity measurements The result of an intermediate verification is a new integrity measurement that captures the semantics of the verification result.

The functions defined in this specification are intended for client-side use but may be useful to server-side verifications, however such use has not been considered in detail at the time of this writing.

The PTS is assumed to be integrity protected by other platform mechanisms including but not limited to protections provided by the OS kernel, drivers, firmware and hardware.

Objects to be protected from unauthorized modification include the platform's persistent image of integrity protected components (configuration data and executables) and in-memory representation of executing components.

2.3 Requirements of IF-PTS

IF-PTS API must meet the following requirements:

- Meet the needs of the TNC architecture

  The API must support all the functions and use cases described in the TNC architecture as they apply to the relationship between the TNCC, IMC and NAR components.

- Efficient

  PTS and IMCs may exchange large messages at times in preparation for establishing a trusted network connection. IF-PTS should support exchange of large PDUs and/or fast IPC channel to minimize latency.

  If PTS operations take a long time to complete, asynchronous interfaces are used to prevent the caller from blocking.

- Interoperable

  IF-PTS converts internal integrity measurement logs into a standard format defined by an XML schema to achieve vendor interoperability. IF-PTS protocol data units are defined using a platform independent PDU format to achieve interoperability between PTS services and clients from different vendors.

- Extensible

  IF-PTS is extensible to allow application specific customization.

- Easy to use and implement

  IF-PTS should be easy to use and implement. It should allow implementers to enhance existing products to support the integrity architecture and integrate legacy code without requiring substantial changes. It should make operations easy for system administrators and end-users. Components of the architecture should interoperate with minimal manual configuration.

- Platform-independence

  IF-PTS must be implementable on a wide variety of platforms. At a minimum Windows, Linux UNIX variants should be supported.

2.4 Interface Assumptions

IF-PTS makes the following assumptions about other components in the system:

- Secure Message Transport

The privacy of PTS supplied integrity data is the responsibility of the caller. For example, TNC Client and TNC Server are assumed to provide a communications tunnel that supports confidentiality.

- Reliable Message Delivery

  IF-PTS PDUs are communicated over an inter-process communications channel that is a reliable byte stream.

- PTS Transitive Trust

  The PTS is assumed to be trusted by other TNC components (IMCs, TNCC and NAR).

2.5 Features Provided by IF-PTS

This section documents the features provided by IF-PTS.

2.5.1 Integrity Scan of Platform Components

The IF-PTS interface supports PTS as an integrity scan agent for TNC components. It supports integrity scanning of virtually any process. However, scans of the TNCC, IMC and NAR are believed to be of particular importance given their role in establishing a trusted network connection.

The PTS itself SHOULD be integrity scanned by an integrity measurement agent that extends the *transitive trust* chain from the platform root of trust for measurement (RTM) to the PTS. IF-PTS interface MUST NOT prevent a client from retrieving a transitive trust chain that includes measurements of PTS, if the PTS service was scanned previously.

2.5.2 XML Formatting

IF-PTS implements interfaces for XML encoding of IMC integrity information in an interoperable, vendor-neutral XML format consistent with TCG defined XML schemas.

2.5.3 Integrity Information Logging

When PTS performs scans of other system components or processes, integrity measurements are stored in the PTS integrity measurement log. Log contents can be viewed externally in the form of a snapshot structure. If the PTS implementation makes use of the TPM, PTS MUST support viewing of PTS log entries in snapshot format.

A PTS client may create, update, delete and prepare integrity logs for transport over a network. A PTS integrity log may be decomposed into multiple snapshots. Snapshots contain a sequence of integrity values and a composite hash for calculating an overall integrity value.

Clients of PTS that generate their own integrity measurements can log them using the PTS by supplying raw measurements as input to IF-PTS and IF-PTS will return them in snapshot format.

When a TPM exists on the platform, the pre-boot integrity log is imported into the PTS integrity measurement log. The pre-boot log is made available externally using the snapshot format.

2.5.4 Digital Signature

Integrity measurements may be signed using a key supplied by the caller. If a TPM is available on the platform, the Quoting Key SHOULD be an Attestation Identity Key, which can be used as part of the Platform Authentication process and the Signing Key SHOULD be certified by an AIK. TPM *attestation identity keys* can be used to *certify* (sign) other signing keys as this provides attestation that the signing keys, if they are nonmigratable or certified migratable keys, reside in the certifying TPM.

If a TPM is available on the platform, then Integrity Reports MAY include TPM quote function results.

3 IF-PTS Implementation Considerations

3.1 Platform Independence

IF-PTS defines a set of messages (PDUs) that can be exchanged over a variety of IPC mechanisms. Named-pipes is the minimum to implement.

3.1.1 Representation of Information

3.1.1.1 Structure Endian Conventions

The minimum to implement in support of this specification is *big endian* format. See section 7.4 for more detail.

3.1.1.2 Byte Packing

All structures MUST be packed on a byte boundary.

3.1.1.3 Lengths

The “Byte” is the unit of length when the length of a parameter is specified.

3.2 PTS Deployment Model

Interoperation with the PTS is achieved through an interface definition (IDL) that defines messages that may be exchanged with the PTS. Processes interact with PTS over a platform specific IPC mechanism. Client vendors can interoperate with any vendor’s PTS at the PDU layer (Figure 3) using the PDU interface over named pipes. If other IPC mechanisms are implemented the PTS_GetCapabilities interface can be used to discover the other IPC mechanisms. Section 7 contains platform specific details needed to setup the IPC channel for various operating systems.

It may be convenient for multiple IMCs operating in the same process address space to share a common library (e.g. DLL) for access to PTS capabilities; see IF-PTS API in Figure 3. The IF-PTS API can be employed in both client and server deployment models, however this specification does not at this time attempt to define the IF-PTS API. PTS services can be exposed through the PDU interface.

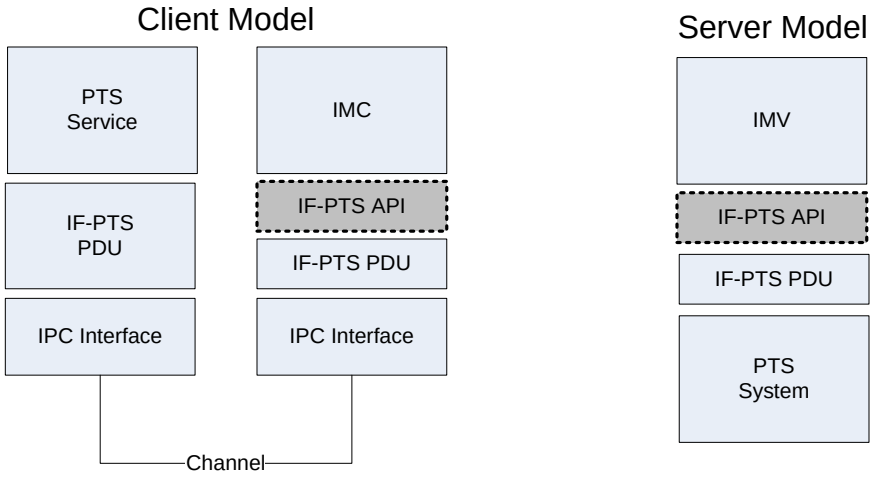


Figure 3 - PTS Deployment Model

3.3 Extensibility

To meet the extensibility requirement defined above, the IF-PTS includes extensibility mechanisms including:

- Interface versioning
- Vendor IDs
- Vendor-specific functions
- XML formatting

3.3.1 Interface Versioning

This document defines version 0 of the IF-PTS API. Future versions may be incompatible due to removing, adding, or changing functions, types, and constants. However, the `PTS_Initialize` function and its associated types and constants will not change so that version incompatibilities can be detected. A PTS can even support multiple versions of the IF-PTS API for maximum compatibility. See section 6.3.1 for details.

3.3.2 Vendor IDs

The IF-PTS API supports several forms of vendor extensions. PTS vendors can define vendor-specific functions and make them available. PTS vendors can define vendor-specific error codes. And vendors of other TNC components can define vendor-specific message types (for the messages sent between them).

In each of these cases, SMI Private Enterprise Numbers are used to provide a separate identifier space for each vendor. IANA provides a registry for SMI Private Enterprise Numbers at http://www.iana.org/assignments/enterprise-numbers. Any organization (including non-profit organizations, governmental bodies, etc.) can obtain one of these numbers. Within this document, SMI Private Enterprise Numbers are known as “vendor IDs”. Vendor ID zero (0) is reserved for identifiers defined by the TNC. For details of how vendor IDs are used to support vendor-specific functions, error codes, and message types, see sections 3.3.3, 5.2.3, 6.1 and 6.2.

3.3.3 Vendor-Specific Functions

The IMC and TNC client MAY extend the IF-PTS API by defining vendor-specific functions that go beyond those described here. A PTS MUST work properly if a vendor-specific function is not implemented by the other party and MUST ignore vendor-specific functions that it does not understand.

Vendor-specific functions MUST specify a vendor ID (see 3.3.2) in the request and response message headers.

The VendorId of zero is used in the normal case for standard ordinals as defined by this specification.

3.3.4 XML Formatting

IF-PTS formats snapshots and integrity reports based on TCG specified XML schemas. For more information see:

- IWG Core Integrity Manifest Schema Specification v1.0
- IWG Snapshot Schema Specification v1.0
- IWG Integrity Report Schema Specification v1.0

3.4 Naming Conventions

To avoid name conflicts, all identifiers in the IF-PTS API have a name that begins with “PTS_”.

Functions described in this document that are to be implemented by a PTS have a name that begins with “PTS_”.

Data structures begin with “PTS_”.

3.5 Threading, Reentrancy and Inter-Process Communication

The PTS is required to be reentrant (able to receive and process a function call even when one is already underway) and should be in a separate process address space from other TNC components.

TNC components will communicate with the PTS through a local inter-process communications (IPC) interface. The default IPC mechanism uses named pipes.

3.6 Types of Messages

Commands are passed between the PTS and other system processes using named pipes. Data structure marshalling is platform specific and therefore intentionally left undefined.

Calling semantics are of two varieties, a request-response or request-response with asynchronous notification. Request commands consist of a request message containing the command ordinal and extensible parameter structure. The request command is followed by a response message flowing in the opposite direction as the request. The response message contains the command ordinal, return value and extensible parameter structure. Asynchronous commands contain a command ordinal and extensible parameter structure with no expectation of an immediate response.

3.7 IPC Resources

Different IPC mechanisms have different conventions for allocating, freeing and recovering IPC resources. This specification intends for PTS implementations to follow such conventions. For example, if a PTS or PTS-IMC process abnormally terminates, the IPC resources are returned to the operating system and PTS / PTS-IMC must recover inconsistent internal state.

Session identifiers negotiated as part of session establishment are temporal. These resources are discarded upon normal and abnormal session termination.

Cookies, allocated as part of asynchronous command invocation, do not persist outside of the session context in which the command was issued. Pending asynchronous events and any associated resources are discarded when the session context is lost. If PTS is holding locks or other system resources pending delivery of an asynchronous notification, these resources are returned to the OS.

Handle resources are freed (not guaranteed to be available) immediately following completion of the final (or terminating) command that used the handle resource and upon termination of the session.

Only values found in snapshot, RIMM and policy structures (such as UUID and ComponentId) will persist between sessions, reset and failures.

3.8 Operational Profiles

IF-PTS is a core specification which defines the capabilities that may be offered in a variety of operational environments. Examples of these environments include a PC client or server where a TPM may or may not be present, a client or server environment which may or may not have a Trusted OS, a mobile phone with or without a TPM, or a secure storage device with trusted computing capabilities. As such, each of these environments will have different security capabilities and PTS requirements. TCG will address these diverse operating environments with PTS-specific profiles which define mandatory and optional capabilities for these corresponding environments.

3.9 TPM PCR Use

When a TPM is present and used as part of the measurement process, PTS requires the use of one resetable TPM PCR for maintaining the integrity of application measurement. A second PCR,

non-resetable by PTS is needed to contain a measurements of the PTS image that becomes part of the platform transitive trust chain that is tied to the platform RTM.

If multiple instances of PTS are created, then each MUST use a different resetable TPM PCR. , or some mechanism must be provided to ensure that PTS measurements have exclusive use of the resetable PCR during application measurement. The code that measures PTS images may extend into a single PCR even when multiple instances of PTS processes are loaded.

IF-PTS RECOMMENDS the use of PCR#22 for measurement of PTS itself. Although PCR#22 SHOULD NOT be resetable, but observe that in the current PC Client Platform Specific specification it is resetable. PCR#23 is used for measurements taken by PTS and SHOULD be resetable by PTS. We observe however that it is resetable by non-PTS parties.

4 Constant Values

This section describes the constants defined in the abstract IF-PTS API.

4.1 Result Codes

Each function in the IF-PTS API returns an error code of type `PTS_Error` to indicate success or reason for failure. Here is the set of standard error codes defined by this specification. Vendor-specific error codes are always permissible using the vendor ID in the command request / response structure. Additional standard error codes may be defined subsequent to the publishing of this specification that would not constitute a change to the negotiated version number at session initialization. The callers of PTS functions MUST be prepared for any function to return any error code. Vendor-specific error codes MUST specify a vendor ID in the response message headers.

If a function returns `PTS_FATAL`, then the TNC component has encountered an unrecoverable error. The PTS SHOULD call `PTS_ Terminate` as soon as possible. The PTS should then take the appropriate action to reset the platform environment if appropriate.

Code	Value
PTS_SUCCESS	0
PTS_FATAL	1
PTS_NOT_INITIALIZED	2
PTS_NO_COMMON_VERSION	3
PTS_UNRECOGNIZED_VENDOR	4
PTS_UNRECOGNIZED_COMMAND	5
PTS_INVALID_COMMAND	6
PTS_INVALID_OPCODE	7
PTS_OPCODE_NOT_DEFINED	8
PTS_COMMAND_NOT_IMPLEMENTED	9
PTS_FEATURE_NOT_IMPLEMENTED	10
PTS_INVALID_SESSION	11
PTS_SESSION_NAME_NOT_FOUND	12
PTS_INVALID_HANDLE	13
PTS_HANDLE_EXIST	14
PTS_PARAMETER_SIZE_MISMATCH	15
PTS_INVALID_TTCHAIN	16
PTS_INVALID_COLLECTOR	17
PTS_UNKNOWN_OWNER	18
PTS_INVALID_PARAMETER	19
PTS_COMPONENT_NOT_FOUND	20
PTS_COMPONENT_REF_EXISTS	21

PTS_REGISTRY_KEY_NOT_FOUND	22
PTS_INVALID_INTERVAL_VALUE	23
PTS_INVALID_ADDRESS	24
PTS_INVALID_SNAPSHOT	25
PTS_SCAN_ABORTED	26
PTS_FILE_SYSTEM_ERROR	27
PTS_SNAPSHOT_NOT_FOUND	28
PTS_DUPLICATE_SNAPSHOT	29
PTS_SNAPSHOT_ACCESS_DENIED	30
PTS_SYNC_SNAPSHOT_NOT_FOUND	31
PTS_INVALID_DESCRIPTOR	32
PTS_REPORT_NOT_FOUND	33
PTS_VERIFY_FAILED	34
PTS_RULE_NOT_FOUND	35
PTS_INVALID_RULE	36
PTS_RULE_NOT_AUTHORIZED	37
PTS_QUOTE_FAILED	38
PTS_SIGN_FAILED	39
PTS_ALREADY_SIGNED	40
PTS_INVALID_PCR_SELECTION	41
PTS_INVALID_KEY	42
PTS_KEY_LENGTH_UNSUPPORTED	43
PTS_CORRUPT_KEY	44
PTS_KEY_NOT_FOUND	45
PTS_UUID_REUSED	46
PTS_INVALID_TPM_LOG	47
PTS_TPM_NOT_FOUND	48
PTS_TPM_OTHER_ERROR	49
PTS_INVALID_XML	50
PTS_INSUFFICIENT_RESOURCES	51
PTS_INVALID_CANONICALIZATION	52
PTS_INVALID_CONFIDENCE	53
PTS_INVALID_PASS_PHRASE	54
PTS_INVALID_FLAGS	55
PTS_DENIED	56

PTS_OS_ERROR	57
PTS_INTERNAL_ERROR	58
PTS_OTHER	59
PTS_DUPLICATE_COOKIE	60
PTS_AUTHENTICATION_FAILURE	61
PTS_INVALID_DIGEST_METHOD	62

4.2 Component Status

This is the set of permissible values for the PTS_ComponentStatus type in this version of the IF-PTS API.

Component Status Value	Value	Definition
PTS_COMPONENT_STATUS_INSTALLED	1	A system component (e.g. TNCC, IMC or other) has been installed.
PTS_COMPONENT_STATUS_UNINSTALLED	2	A system component has not been installed successfully.

4.3 PTS Command Ordinals

The PTS uses an IPC style interface where each function consists of a command having both a request and response message. Input parameters are passed over the request message. Output parameters including the result code are passed over the response message. Each command has a unique command ordinal.

Name	Value	Description
PTS_INITIALIZE	**0**	
PTS_TERMINATE	**1**	
PTS_COMPONENT_SCAN	**2**	
PTS_COMPONENT_SCAN_COMPLETE	**3**	Asynchronous
PTS_COMPONENT_LOCKED	**4**	
PTS_COMPONENT_UNLOCKED	**5**	Asynchronous
PTS_SNAPSHOT_SYNC	**6**	
PTS_SNAPSHOT_SYNC_COMPLETE	**7**	Asynchronous
PTS_SNAPSHOT_VERIFY	**8**	
PTS_SNAPSHOT_SIGN	**9**	
PTS_SNAPSHOT_CREATE	**10**	
PTS_SNAPSHOT_DELETE	**11**	
PTS_SNAPSHOT_IMPORT	**12**	
PTS_SNAPSHOT_EXPORT	**13**	

PTS_SNAPSHOT_GET_PROPERTIES	**14**	
PTS_SNAPSHOT_OPEN	**15**	
PTS_SNAPSHOT_CLOSE	**16**	
PTS_SNAPSHOT_UPDATE_COMPONENTID	**17**	
PTS_SNAPSHOT_UPDATE_SUBCOMPONENTS	**18**	
PTS_SNAPSHOT_UPDATE_ASSERTIONS	**19**	
PTS_SNAPSHOT_UPDATE_INTEGRITY_VALUES	**20**	
PTS_SNAPSHOT_UPDATE_COLLECTOR	**21**	
PTS_REPORT_CREATE	**22**	
PTS_REPORT_DELETE	**23**	
PTS_REPORT_SPECIFY	**24**	
PTS_REPORT_GENERATE	**25**	
PTS_REPORT_GET_PROPERTIES	**26**	
PTS_REPORT_VERIFY	**27**	
PTS_REGISTER_RULE	**28**	
PTS_UNREGISTER_RULE	**29**	
PTS_LIST_RULE	**30**	
PTS_CONFIGURE_PCR	**31**	
PTS_REGISTER_QUOTE_KEY	**32**	
PTS_UNREGISTER_QUOTE_KEY	**33**	
PTS_LIST_QUOTE_KEYS	**34**	
PTS_REGISTER_SIGNING_KEY	**35**	
PTS_UNREGISTER_SIGNING_KEY	**36**	
PTS_LIST_SIGNING_KEYS	**37**	
PTS_GET_CAPABILITIES	**38**	
PTS_LIST_SUPPORTED_ALGORITHMS	**39**	
PTS_REGISTER_VERIFY_KEY	**40**	
PTS_DEREGISTER_VERIFY_KEY	**41**	
PTS_LIST_VERIFY_KEYS	**42**	
PTS_GET_COOKIE	**43**	

4.4 Snapshot Flags

These are bit values that represent the modes in which a snapshot can be opened.

Access Flag	Value	Description
PTS_ACCESS_WRITE	0x0001	Open for write access

Share Flag	Value	Description
PTS_EXCLUSIVE	0x0000	access in exclusive mode (no sharing)
PTS_SHARE	0x0001	Allow others to write.

4.5 Miscellaneous Constants

Constant Definition	Value	Definition
PTS_VERSION_0	0	The version of IF-PTS API defined here
MAXINT	0xFFFFFFFF	Maximum value for a PTS_UInt32
UUIDSIZE	16	A UUID is a 128 bit object

4.5.1 Version Numbers

As noted in section 3.3.1, this specification defines version 0 of the TNC IF-PTS API. Future versions of this specification will define other version numbers. See section 6.3.1 for a description of how version numbers are handled.

5 Data Structures

This section describes the data types defined and used in the abstract IF-PTS API.

5.1 Basic Types

These are the basic data types used by the IF-PTS API. They are defined in a platform-independent and language-independent manner to meet the requirements described in this section. Consult section 7 to see how these types are defined for a particular platform and language.

Type	Definition
PTS_Byte	8 bit octet
PTS_UInt16	Unsigned integer of 16 bits
PTS_UInt32	Unsigned integer of 32 bits
PTS_UInt64	Unsigned integer of 64 bits
PTS_UINT	Interpreted as PTS_UIntX and is specified in the Platform specific section 7. Unsigned integer of X bits. (This is used as a last resort when it isn't possible to define word size explicitly).
PTS_Bool	Octet sized enumeration where 0=FALSE and 1=TRUE
PTS_VoidPtr	Unsigned pointer to void – the size of this pointer is defined by a platform specific specification. E.g. on a 16-bit platform, it is an unsigned integer of 16 bits; on a 32-bit platform, it is an unsigned integer of 32 bits; on a 64-bit platform, it is an unsigned integer of 64 bits.

To declare an array of structures, this document uses the convention of appending brackets "[]" to a structure. For example, an array of rules is declared as follows: "rules[]".

5.2 Simple Derived Types

These types are defined in terms of the more basic ones defined in section 5.1 they are described in the following subsections.

5.2.1 PTS_AlgorithmId

```
typedef PTS_String PTS_AlgorithmId; // Algorithm URI
```

`PTS_AlgorithmId` holds a URI identifying a particular algorithm.

5.2.2 PTS_ComponentStatus

```
typedef PTS_UInt32 PTS_ComponentStatus;
```

PTS_ComponentStatus hold a bit flag of component measurement status values.

5.2.3 PTS_Cookie

```
typedef PTS_UInt64 PTS_Cookie;
```

A cookie is used by clients to correlate asynchronous function calls.

5.2.4 PTS_Error

```
typedef PTS_UInt32 PTS_Error;
```

Each function in the IF-PTS API returns an error code of type `PTS_Error` to indicate success or the reason for failure.

Clients MUST be prepared for any function to return vendor specific error codes. The VendorId must be checked before checking the error code. Vendor-specific error codes are always permissible and new standard error codes may be defined without changing the version number of the IF-PTS interface.

5.2.5 PTS_Handle

```
typedef PTS_UInt64 PTS_Handle;
```

A handle is used to identify the context for report generation.

5.2.6 PTS_PcrId

```
typedef PTS_UInt32 PTS_PcrId;
```

An identifier corresponding to a PCR register.

5.2.7 PTS_SessionName

```
typedef PTS_String PTS_SessionName;
```

For each client IPC instance that interacts with the PTS, a unique session is created. The session name is used by the client to maintain session context with the PTS. Session names are opaque to the client. Client code MUST NOT anticipate any particular naming convention will be followed. SessionName is only used for testing uniqueness.

5.2.8 PTS_SnapshotDescriptor

```
typedef PTS_UInt32 PTS_SnapshotDescriptor;
```

Descriptor to a snapshot that is open for reading or update. Descriptor values are allocated by PTS and are opaque to the client.

5.2.9 PTS_SnapshotFlags

```
typedef struct
{
      PTS_UInt16 access;
      PTS_UInt16 share;
}PTS_SnapshotFlags;
```

A snapshot can be opened with different modes.

See Section 4.4 for more description.

5.2.10 PTS_SnapshotId

```
typedef PTS_UUID  SnapshotId;
```

A unique identifier of a snapshot structure.

5.2.11 PTS_UUID

```
typedef struct {
   PTS_Byte[UUIDSIZE]   idVal;
} PTS_UUID;
```

Description

Name	Description
idVal	The octets containing a universally unique identifier formatted in accordance with RFC 4122.

Unique vendor identifiers expressed as a Universal Unique Identifier (UUID) formatted in accordance with RFC4122. Conversion from UUID string format to UUID binary representation SHOULD be carried out according to RFC4122.

5.2.12 PTS_VariableLengthDataPtr

```
typedef struct {
      PTS_UInt32  offset;
      PTS_UInt32  length;
}PTS_VariableLengthDataPtr;
```

PTS_VariableLengthDataPtr is a 32-bit offset that points to a buffer containing strings.

5.2.13 PTS_Version

```
typedef PTS_UInt32 PTS_Version;
```

The PTS_Version describes the API version. See sections 3.3.1 and 6.3.1.

5.3 Complex Data Types

5.3.1 PTS_AddByCollector

```
typedef struct {
   PTS_ComponentId collector;
   PTS_UInt32     treeDepth;
} PTS_AddByCollector;
```

Description

Name	Description
collector	The snapshot is selected if its collector componentId matches *collector*.
treeDepth	If non-zero, the sub tree for each selected snapshot is included to the depth specified by treeDepth.

	A depth of MAXINT will select the entire tree regardless of depth.

This structure is used to add components to an integrity report. Tree depth refers to the number of layers of sub-components that may be nested under the given snapshot. Refer to the *IWG Reference Architecture Part II* and the *IWG Core Integrity Manifest Schema* specifications for additional details regarding Snapshot construction and architecture.

5.3.2 PTS_AddByComponent

```
typedef struct {
   PTS_ComponentId      componentId;
   PTS_Bool             partialMatchFlag;
   PTS_UInt32           treeDepth;
} PTS_AddByComponent;
```

Description

Name	Description
componentId	The component identifier used to locate a snapshot. ComponentId defined in section 5.3.8.
partialMatchFlag	If TRUE, any element/attribute of the componentId that matches corresponding element/attribute in a collection of snapshots will be selected. I.e. logical OR is applied to matching fields to select the snapshot. E.g. if componentId.ModelName=”XYZ” and componentId.VersionString=”123” and snapshot1.ModelName=”XYZ” and snapshot1.VersionString=”789”; then snapshot1 will be selected. If FALSE, each element in componentId must exactly match corresponding elements/attributes in a snapshot in order to be selected. I.e. logical AND is applied to matching fields to select the snapshot. E.g. if componentId.ModelName=”XYZ” and componentId.VersionString=”123” and snapshot1.ModelName=”XYZ” and snapshot1.VersionString=”789”; then snapshot1 will NOT be selected.
treeDepth	If non-zero, the component subtree, identified by componentId will be used to select snapshots for inclusion in the report. The depth specified by treeDepth is used to terminate selection of nested sub-components at treeDepth layer. A depth of MAXINT will select the entire tree regardless of depth.

This structure is used to add components to an integrity report.

5.3.3 PTS_AddByOwner

```
typedef struct {
   PTS_UUID       ownerId;
   PTS_UInt32     treeDepth;
} PTS_AddbyOwner;
```

Description

Name	Description

ownerId	The snapshot is selected if its owner ID matches *ownerId*.
treeDepth	If non-zero, the subtree for each selected snapshot is included to the depth specified by treeDepth. A depth of MAXINT will select the entire tree regardless of depth.

This structure is used to add components to an integrity report.

5.3.4 PTS_AddByPcr

```
typedef struct {
   PTS_PcrBitmask pcrSelection;
   PTS_UInt32     treeDepth;
} PTS_AddByPcr;
```

Description

Name	Description
pcrSelection	The snapshot is selected if it corresponds to a TPM PCR specified in *pcrSelection*. PTS_PcrBitmask defined in section 5.3.14.
treeDepth	If non-zero, the subtree for each selected snapshot is included to the depth specified by treeDepth. A depth of MAXINT will select the entire tree regardless of depth.

This structure is used to add components to an integrity report.

5.3.5 PTS_AddByTrustChain

```
typedef struct {
   PTS_UInt32     treeDepth;
   PTS_SnapshotId pathTerminator;
} PTS_AddByTrustChain;
```

Description

Name	Description
pathTerminator	The snapshotId for the terminating link in a transitive trust chain leading to the Root-of-trust for Measurement (RTM).
treeDepth	If non-zero, the subtree for each snapshot of the transitive trust chain is included to the depth specified by treeDepth. A depth of MAXINT will select the entire tree regardless of depth.

This structure is used to add components to an integrity report.

5.3.6 PTS_AssertionsInfo

```
typedef struct {
   PTS_UInt32                numAssertions;    // number of assertions
in the list
   PTS_VariableLengthDataPtr  assertionList; // the first assertion in
the list – each assertion is of type PTS_String
   PTS_VariableLengthDataArea assertionData;    // variable length data
} PTS_AssertionsInfo;
```

Description

A list of text blobs containing XML formatted assertions. numAssertions indicates the number of entries in the assertionsList. The PTS_String structure contains XML formatted assertionInfo. PTS treats assertionList values as opaque data. The assertionList entries are offsets into assertionData that contains an array of variable length strings.

PTS copies the assertion text into a snapshot assertionInfo field.

5.3.7 PTS_Capability

```
typedef struct {
      PTS_UInt32  vendorId;
      PTS_UInt32  commandOrdinal;
      PTS_UInt32  implementationStatus;   // feature bitmask
} PTS_Capability;
```

A tuple identifying a vendorId, a command ordinal and the implementation status for the specified commandOrdinal.

5.3.8 PTS_ComponentId

```
typedef struct {
   PTS_VariableLengthDataPtr  vendor;           // PTS_Vendor: vendor
                                                   or manufacturer
   PTS_VariableLengthDataPtr  simpleName;       // PTS_String: simple
                                                   name
   PTS_VariableLengthDataPtr  modelName;        // PTS_String: model
                                                   name
   PTS_VariableLengthDataPtr  modelNumber;      // PTS_String: model #
   PTS_VariableLengthDataPtr  modelSerialNumber;// PTS_String: model
                                                   serial number
   PTS_VariableLengthDataPtr  modelSystemClass; // PTS_String: model
                                                   System class
   PTS_Version                majorVersion;     // PTS_Version: major
                                                   version
   PTS_Version                minorVersion;     // PTS_Version: minor
                                                   version
   PTS_UInt32                 buildNumber;      // build or series
                                                   number
   PTS_VariableLengthDataPtr  versionString;    // PTS_String: string-
                                                   ified version
   PTS_VariableLengthDataPtr  patchLevel;       // PTS_String: patch
                                                   level
   PTS_VariableLengthDataPtr  discretePatches;  // PTS_String: white
                                                   space delimited
                                                   discrete patch names
   PTS_DateTime               buildDate;        // date and time of
                                                   release
   PTS_VariableLengthDataArea dataBlock;        // variable length data
} PTS_ComponentId;
```

Description

The component ID is a collection of attributes that identifies a software or hardware release. There may be multiple instances of the same component identifier.

5.3.9 PTS_DateTime

```
typedef struct {
   PTS_UInt32     sec;  // seconds range: 0-59
   PTS_UInt32     min;  // minutes range: 0-59
   PTS_UInt32     hour; // hour range: 0-23
   PTS_UInt32     mday; // day of the month range: 1-31
   PTS_UInt32     mon;  // month range: 1-12
   PTS_UInt32     year; // year range: 0-MAXINT
   PTS_UInt32     wday; // day of the week range: 0-6 (0=Sun, 6=Sat)
   PTS_UInt32     yday; // day of the year range: 0-365
   PTS_Bool       isDst;      // true if daylight savings
} PTS_DateTime;
```

Description

The date and time expressed in GMT time zone.

5.3.10 PTS_IntegrityReport

```
typedef struct {
      PTS_UInt32  reportSize;
      PTS_Byte    reportData;
} PTS_IntegrityReport;
```

Description

Structure containing XML formatted snapshots.

5.3.11 PTS_Key

```
typedef struct {
      PTS_UInt32  keyLength;
      PTS_UUID    keyId;
} PTS_Key;
```

Description

A UUID that identifies a keys held in a key storage token or device (e.g. TPM). PTS_Keys are used for signing / verifying or encrypting / decrypting (e.g. *sealing* / *binding*). A NIL / NULL key conforms to RFC 4122 definition for “nil” keys (e.g. 00000000-0000-0000-0000-000000000000)

5.3.12 PTS_MemSegment

```
typedef struct {
   PTS_UINT       size;
   PTS_VoidPtr    addr;
} PTS_MemSegment;
```

Description

Name	Description
size	The size in bytes of a memory segment starting at *addr* memory address. Interpreted as UIntX where X is defined in a platform specific binding.
addr	A memory address.

The beginning of a memory segment is identified by *addr* and the size of the memory segment is contained in *size*. This structure can be used to identify a memory segment to be scanned.

5.3.13 PTS_MemSegments

```
typedef struct {
   PTS_UInt32     numSegments; // number of PTS_MemSegment structures
   PTS_UInt64     offset; // VariableLengthDataArea of first structure
   PTS_UInt32     length; // in bytes of all PTS_MemSegment structures
}PTS_MemSegments;
```

Points to a list of PTS_MemSegment structures contained in a PTS_VariableLengthDataArea.

5.3.14 PTS_PcrBitmask

```
typedef struct {
      PTS_UInt32 sizeOfSelect;
      PTS_UInt8 pcrSelect[]; //variable length array of octets
} PTS_PcrBitmask;
```

Description

Name	Description
sizeOfSelect	The size in bytes of the pcrSelect array
pcrSelect	A bitmask of PCR selections where a non-zero bit indicates selection

pcrSelect is a contiguous bit map that shows which PCRs are selected. Each byte is a bitmask representing 8 PCRs. Byte 0 indicates PCRs 0-7, byte 1 (8-15) and so on.

When an individual bit is 1 the indicated PCR is selected. If 0 the PCR is not selected.

Byte 0

7	6	5	4	3	2	1	0

Byte 1

F	E	D	C	B	A	9	8

Byte 2

17	16	15	14	13	12	11	10

Etc…

5.3.15 PTS_ReportProperties

```
typedef struct {
      PTS_UInt64  size;       // in bytes
      PTS_Bool    isSigned;   // true if digital signature exists
      PTS_Bool    isQuoted;   // true if TPM Quote exists
      PTS_PcrBitmask    pcrs; // pcr(s) used in the integrity report
} PTS_SnapshotProperties;
```

Description

This structure contains properties of an integrity report.

5.3.16 PTS_SignerInfo

```
typedef struct {
      PTS_VariableLengthDataPtr     canonicalizationAlg;
                                                  // PTS_AlgorithmId
      PTS_UInt32                    confidenceValue;
      PTS_UInt32                    confidenceBase;
      PTS_VariableLengthDataPtr     confidenceUri;
      PTS_Key                       signingKey;
      PTS_VariableLengthDataArea    data;
} PTS_SignerInfo;
```

Description

Name	Description
canonicalizationAlg	Specifies the algorithm in SignerInfo used to remove whitespace prior to signing.
confidenceValue	Confidence level that collected measurements are accurate
confidenceBase	Divisor applied to confidenceValue; Must be greater than 0.
confidenceUri	URI describing confidence value calculation method
signingKey	Reference to the signing key

5.3.17 PTS_SnapshotProperties

```
typedef struct {
      PTS_UUID    owner;
      PTS_UInt64  size;       // in bytes
      PTS_Bool    isSigned;   // true if digital signature exists
      PTS_Bool    isSynced;   // true if synced to a TPM PCR
      PTS_PcrId   syncPCR;    // pcr number to which snapshot is synced
} PTS_SnapshotProperties;
```

Description

This structure contains properties of a snapshot.

5.3.18 PTS_String

```
typedef struct {
   PTS_UInt32     size;
   PTS_Byte       stringData;
} PTS_String;
```

Description

PTS_String is used for variable length strings that do not contain external pointer references. The *size* value indicates the number of UTF8 characters in the *stringData*. Strings do not require NULL termination.

5.3.19 PTS_ValuesInfo

```
typedef struct {
   PTS_VariableLengthDataPtr  id;         // PTS_UInt32
   PTS_VariableLengthDataPtr  name;       // PTS_String
   PTS_VariableLengthDataPtr  objectRef;  // PTS_String
   PTS_VariableLengthDataPtr  type;       // PTS_String
   PTS_VariableLengthDataPtr  digestMethod   // PTS_AlgorithmId
   PTS_VariableLengthDataPtr  transformMethod; // PTS_AlgorithmId
   PTS_VariableLengthDataPtr  dataToHash; // PTS_String
   PTS_VariableLengthDataPtr  digest;     // PTS_String
   PTS_VariableLengthDataArea data;
} PTS_ValuesInfo;
```

Description

Name	Description
id	Supplied Identifier for this integrity value – if the value is nil, then no Id is supplied (Id is an optional input). In the Simple Object Schema, this is the Object.Id.
name	Name of the integrity value – as Name is optional, if it is nil, then the size will be 0 bytes.
objectRef	Reference to the raw data – this is a URI. As ObjectRef is optional, if it is nil, then the size will be 0 bytes.
type	Type of integrity value – As Type is optional, if it is nil, then the size will be 0 bytes.
digestMethod	The URI of the Digest Method. If Data to Hash is used, then this is the digest method to be used by the TPM to calculate the digest over the raw data to hash. The caller should call GetCapabilites to ensure a Digest Method supported by PTS is input.
transformMethod	The URI of the Transform Method applied to the data prior to input to PTS. If nil, then the size will be 0 bytes.
dataToHash	The opaque data to hash – if this is not nil, then digest is nil
digest	The actual digest value – if this is not nil, then DataToHash is nil
data	The data buffer containing packed parameters.

This defines a structure that contains integrity measurements. See also TCG Core Integrity Schema.

5.3.20 PTS_VariableLengthDataArea

```
typedef struct {
   PTS_UInt32     blockSize;  // size in bytes of dataBlock
   PTS_Byte       dataBlock;  // blob containing variable length data
} PTS_VariableLengthDataArea;
```

Description

A variable size block of data in octets.

5.3.21 PTS_Vendor

```
typedef struct {
```

```
   PTS_UInt32     tcgVendorId;
   PTS_UInt32     smiVendorId;
   PTS_UUID       vendorGUID;       // Vendor supplied UUID
   PTS_String     vendorName;
} PTS_Vendor;
```

Description

Name	Description
tcgVendorId	Vendor ID supplied by TCG
smiVendorId	SMI Private Enterprise Number
vendorGUID	UUID that identifies the vendor if the vendor lacks SMI or TCG vendor IDs
nameLen	Length of the vendorName
vendorName	Vendor Name string

Unique vendor identifiers

6 Commands

PTS functions are realized as messages communicated over an inter-process communications channel. Nevertheless a functional definition is provided for clarity and consistency across IPC mechanisms and marshalling techniques.

6.1 Request Message

All request messages are constructed using a PTS_Message_Request structure. The command ordinal and size are used to parse the data portion of the message. The ordinal determines which data type should be used to perform a cast operation over the byte array.

```
typedef struct {
      PTS_UInt32 vendorId;          // SMI Private Enterprise Number
      PTS_UInt32 command;           // command ordinal value
      PTS_UInt32 size;              // total size of request “data”
      PTS_Byte   params;            // first byte of the parameter list
}PTS_Message_Request;
```

Description

Parameter	Description
vendorId	SMI private enterprise number indicates a vendor specific extension. Zero (0x0) indicates command ordinals defined by this specification. This is the PTS VendorId – for vendor-specific commands.
command	Command ordinal for a particular request message. Command ordinals may be reused in response messages (see 6.2).
size	The size in bytes of the remaining message structure.
params	The first byte in the remainder of the request message. The command ordinal indicates which data structures are used to interpret the remaining bytes and offsets.

6.2 Response Message

Response messages are constructed using a PTS_Message_Response structure. The command ordinal is the same as that used for the request. Response messages include the result code resulting from execution of the request. If there is an error in the Response Message, there is no mechanism for the calling application to communicate the error condition to PTS. The error codes defined in this generic Response Message apply to all Response Messages and are not explicitly listed as error codes in the subsequent commands.

```
typedef struct {
      PTS_UInt32  vendorId;         // SMI Private Enterprise Number
      PTS_UInt32  command;          // command ordinal value
      PTS_Error   errCode;          // result code
      PTS_UInt32  size;             // total size of response “data”
      PTS_Byte    params;           // first byte of the parameter list
}PTS_Message_Response;
```

Description

Parameter	Description

vendorId	SMI private enterprise number indicates a vendor specific extension. Zero (0x0) indicates command ordinals defined by this specification.
command	Command ordinal.
errCode	The result code from execution of a previous request.
size	The size in bytes of the remaining message structure.
params	The first byte in the remainder of the request message. The command ordinal indicates which data structures are used to interpret the remaining bytes and offsets.

Result Codes	Description
PTS_SUCCESS	Success
PTS_OTHER	Unspecified non-fatal error
PTS_FATAL	Unspecified fatal error
PTS_INVALID_COMMAND	Invalid Command Sequence / Illegal Command State
PTS_COMMAND_NOT_IMPLEMENTED	Requested command not implemented
PTS_PARAMETER_SIZE_MISMATCH	Parameter size is incorrect
PTS_UNRECOGNIZED_VENDOR	Vendor not recognized by PTS
PTS_UNRECOGNIZED_COMMAND	Command does not exist
PTS_INVALID_PARAMETER	Relating to the Parameter List
PTS_INSUFFICIENT_RESOURCES	Insufficient resources to service request
PTS_OS_ERROR	Operating system error

Asynchronous response messages are messages created by PTS in response to an earlier message request that required processing time to complete. In this case PTS responds to the original message request with an immediate message response, performs any requisite processing, and then responds with an appropriate asynchronous message response. Asynchronous response messages are also constructed using a PTS_Message_Response structure. The command ordinal is not the same as that used in the request, but rather the command ordinal of the PTS asynchronous command. Asynchronous response messages include the result code resulting from execution of the asynchronous command request. If there is an error in the Response Message, there is no mechanism for the calling application to communicate the error condition to PTS. The error codes defined in this generic Response Message apply to all Response Messages and are not explicitly listed as error codes in the subsequent commands.

6.3 PTS Initialization Commands

6.3.1 PTS_Initialize

```
typedef struct {
      PTS_UUID clientId,            // caller supplied owner info
      PTS_Version minVersion,       // min supported interface
      PTS_Version maxVersion        // max supported interface
```

```
} PTS_Initialize_Request;

typedef struct {
      PTS_Version actualVersion,          // selected interface version
      PTS_SessionName sessionName         // name of this session
} PTS_Initialize_Response;
```

Description

This function is implemented by the PTS and called by client processes seeking PTS services.

A platform component calls this function to initialize the PTS and agree on the API version number to be used.

A well-known IPC channel (e.g. named pipe) is used to send the PTS_Initialize command. Thereafter all other PTS commands will use IPC resources that are associated with a PTS session identified by *sessionName*. Unless it is clear from the context of *sessionName* then the same IPC mechanism SHOULD be used.

The *sessionName* SHOULD be used when constructing an IPC channel, for example naming pipe resources in addition to the default named pipe. The session name is composed of the *clientId* and a unique session identifier chosen by the PTS. The algorithm for composing *sessionName* is unspecified.

If clientId has already been used to initialize a concurrent session, PTS_Initialize will fail and the caller should select a different value for clientId.

Input Parameters	Description
`clientId`	A caller defined unique identifier (e.g. UUID). PTS will use clientId to establish a snapshot owner. The clientId may persist beyond command session lifetimes and system reset allowing snapshots to be re-associated with the caller across multiple command sessions. ClientId = OwnerId
`minVersion`	Minimum version supported by the calling local process
`maxVersion`	Maximum version supported by the calling local process

Output Parameters	Description
`actualVersion`	Actual version selected by PTS
`sessionName`	A session identifier

Error Codes	Condition
`PTS_NO_COMMON_VERSION`	No common IF-PTS API version between PTS and other TNC components
`PTS_NOT_INITIALIZED`	PTS_Initialize failure
`PTS_UUID_REUSED`	The supplied UUID has already been used to uniquely identify another PTS structure

6.3.2 PTS_Terminate

```
typedef struct {
                  // the current session
} PTS_Terminate_Request;
```

Description

This function is implemented by the PTS and is called by a client process.

This function requests the PTS to terminate the current session and to free allocated resources.

Error Codes	Condition
PTS_INVALID_SESSION	Invalid session ID or session name

6.4 Integrity Measurement and Verification Commands

6.4.1 PTS_ComponentScan

```
typedef struct {
      PTS_Cookie                    cookie;
      PTS_VariableLengthDataPtr     componentId; //PTS_ComponentId
      PTS_String                    processName;
      PTS_MemSegments               segments;
      PTS_VariableLengthDataPtr     componentRegistryPath; //PTS_String
      PTS_UInt32                    interval;
      PTS_Uint32                    scanDepth;
      PTS_Bool                      doVerify;
      PTS_Bool                      includeMeasValues; // if doVerify
      PTS_SnapshotId                snapshotId;
      PTS_Uint32                    numRules;
      PTS_UUID                      rules[];
      PTS_VariableLengthDataArea    data;
} PTS_ComponentScan_Request;

typedef struct {
      PTS_SnapshotId outputSnapshotId;
} PTS_ComponentScan_Response;
```

Description

This command is implemented by the PTS and is called by the client process.

PTS_ComponentScan can be used to scan software components found on the file system AND to scan the memory of a component as a running process.

If processName is specified, the component identified by componentId will be scanned in memory. Memory segments are used to identify specific code segments that are to be scanned.

If componentRegistryKey is specified, the component image on disk is scanned. PTS is expected to read system registry information to identify and locate images that are included in the scan. The componentRegistryPath is used to help locate where the component files were installed.

PTS will also use Reference Integrity Measurement Manifest (RIMM) structures to guide the measurement process and in constructing a list of objects included in scans.

The PTS_RegisterRule command may be used to provision PTS with RIMM structures and policy statements that are applied during a verification check. Identifiers for registered rules are provided via the command interface to indicate applicable rules.

If the doVerify flag is TRUE. This command will also verify measurements collected by PTS according to reference measurements contained in RIMM structures.

PTS checks the image using componentId to locate applicable RIMMs.

An integrity **check** (or verification) differs from an integrity **scan** in that integrity checking applies rules that allow PTS to determine whether a computed integrity value is acceptable. The results of the check in the outputSnapshotId include:

- Scan result (e.g. success / failure)
- Rule or policy that was applied

If a memory image scan is requested, then after responding with a PTS_ComponentScan_Response message, PTS will subsequently send a PTS_ComponentScanComplete_Async_Response message after the scan has finished.

If a component disk image scan is requested, then after responding with a PTS_ComponentScan_Response message, PTS will subsequently send a PTS_ComponentLocked_Response message and a PTS_ComponentUnlocked_Async_Response message.

If n in-memory image scan and a component disk image scan are both requested, then after responding with a PTS_ComponentScan_Response message PTS will subsequently send a PTS_ComponentScanComplete_Async_Response message, a PTS_ComponentLocked_Response message and a PTS_ComponentUnlocked_Async_Response message.

The client application is able to find valid componentIds by referencing RIMM structures or a configuration database that is RIMM aware.

Input Parameters	**Description**
`Cookie`	A caller supplied cookie for coordination of component scan context responses
`componentId`	Identifies the component to be scanned
`processName`	Text process identifier that is used to find running process(es) included in the memory scan
`Segments`	segments are the memory addresses of memory sections to be scanned. If MemSegments.NumSegments = 0, no component memory scan is requested.
`componentRegistryPath`	If non-NULL, componentRegistryPath contains a registry pathname where objects associated with ComponentId can be found. This may be a RPM in Linux or a package database in SysV Unix or a registry database in Windows.
`Interval`	If non-zero, the scan operation is repeated after interval seconds has expired. If interval is zero, the scan is performed once. Otherwise the scan will be repeated indefinitely as long as the session exists.

scanDepth	ScanDepth specifies the number of layers of Components to be traversed in the scan. A scanDept of zero (0) does not traverse a lower layer. Lower layer snapshot structures may be generated as needed to contain scan results. A scanDepth value of MAXINT will traverse to component leaf nodes.
doVerify	If TRUE, scan results are compared with reference measurements, contained in RIMM structures identified by ComponentId, and against XACML policies. The evaluates reference measurements and XACML policies referenced by rules. PTS will include in the snapshot structure AssertionInfo the result of the verification, scan depth and identify the RIMM/policy structure that was applied (e.g. RIMM.xs:ID).
includeMeasValues	The includeMeasValues flag is only evaluated if doVerify is TRUE. If includeMeasValues is TRUE, include all measurements values in the outputSnapshotId in addition to the verification results. If includeMeasValues is FALSE, include the verification results in the outputSnapshotId, but do not include the measurement values.
snapshotId	Identifies snapshot used to store scan results. PTS will be the measuring entity. If snapshotId is an RFC 4122 “nil”, the PTS will create a snapshot structure. If scanDepth is >0 then sub-components identified by componentId will be scanned to scanDepth level of nesting and sub-snapshots will be generated as needed to contain scan results. If verification is also performed then the verification results are captured as snapshot assertions.
rules	Identifies a particular reference measurements / rules to be applied when computing measurements and verifying them. (e.g. RIMM.xs:ID). If numRules is 0 and doVerify is TRUE, a best effort attempt will be made to locate an applicable rule using ComponentId. If numRules is 0 and doVerify is FALSE, then no integrity check is requested. If numRules is non-zero and doVerify is TRUE, then only the specified rules are used during verification. If numRules is non-zero and doVerify is FALSE, then rules are ignored.

Output Parameters	**Description**
outputSnapshotId	Snapshot ID of the snapshot containing the results of the measurement process. If the doVerify flag is TRUE, it contains a record of the verification

	actions. The snapshot assertions contain the results values. The contents schema selected is PTS implementation specific.

Error Codes	Condition
PTS_COMPONENT_NOT_FOUND	The component specified by ComponentId or Registry Key could not be found
PTS_SNAPSHOT_NOT_FOUND	A snapshot corresponding to the supplied snapshotId could not be found
PTS_REGISTRY_KEY_NOT_FOUND	Registry objects not found
PTS_FEATURE_NOT_IMPLEMENTED	Memory scan not supported
PTS_ILLEGAL_ADDRESS	Component address is illegal
PTS_INVALID_INTERVAL_VALUE	Interval value not supported
PTS_VERIFY_FAILED	If doVerify flag and the measured values differ from reference measurements.
PTS_RULE_NOT_FOUND	The referenced rule or RIMM could not be found
PTS_INVALID_DIGEST_METHOD	The digest method used in the RIMM is not supported
PTS_VERIFY_NOT_SUPPORTED	Integrity check functionality is not supported in the PTS implementation

6.4.2 PTS_ComponentScanComplete

```
typedef struct {
      PTS_Cookie cookie,
      PTS_ComponentStatus componentStatus,
      PTS_SnapshotId snapshotId
} PTS_ComponentScanComplete_Async_Response;
```

Description

This command is originated by the PTS in response to the client call to PTS_ComponentScan.

This command notifies the completion of a PTS in-memory scan of a component (e.g. IMC, TNCC etc…). A snapshot structure updated with the scan results is referenced by snapshotId.

This command will be issued following a PTS_ComponentScan.

Output Parameters	Description
cookie	A cookie to a component scan context
componentStatus	The status of the component according to the PTS
snapshotId	The snapshot identifier of the scan results

Error Codes	Condition
PTS_SCAN_ABORTED	Memory scan aborted prior to completion

PTS_SNAPSHOT_NOT_FOUND	A snapshot corresponding to the supplied snapshotId could not be found
PTS_VERIFY_FAILED	Scan did not verify against RIMM
PTS_SNAPSHOT_NOT_FOUND	Snapshot ID does not exist

6.4.3 PTS_ComponentLocked

```
typedef struct {
      PTS_Cookie cookie
} PTS_ComponentLocked_AsyncResponse;
```

Description

This command is originated by the PTS in response to the client call to PTS_ComponentScan.

This function notifies the intent of the PTS to do a file scan of a component (see 6.4.1). Files SHOULD be locked by the PTS prior to performing a file scan operation. The PTS will perform the scan subsequent to notification of the caller using this function.

Output Parameters	Description
cookie	A cookie to a file scan context

Error Codes	Condition
PTS_SCAN_ABORTED	File scan aborted prior to completion
PTS_COMPONENT_NOT_FOUND	The component specified by ComponentId could not be found
PTS_SNAPSHOT_NOT_FOUND	A snapshot corresponding to the supplied snapshotId could not be found
PTS_REGISTRY_KEY_NOT_FOUND	Registry objects not found

6.4.4 PTS_ComponentUnlocked

```
typedef struct {
      PTS_Cookie cookie;
      PTS_ComponentStatus componentStatus;
      PTS_SnapshotId snapshotId;
} PTS_ComponentUnlocked_Async_Response;
```

Description

This command is originated by the PTS in response to the client call to PTS_ComponentScan.

This function notifies the completion of a PTS file scan of a component (see 6.4.1). A snapshot structure, if supplied, is updated with the scan results.

Output Parameters	Description
cookie	A cookie to a file scan context
componentStatus	The status of the component according to the PTS

snapshotId	The snapshot identifier of the scan results

Error Codes	Condition
PTS_SCAN_ABORTED	File scan aborted prior to completion
PTS_VERIFY_FAILED	Scan did not verify against RIMM
PTS_SNAPSHOT_NOT_FOUND	Snapshot ID does not exist

6.4.5 PTS_SnapshotSync

```
typedef struct {
      PTS_SnapshotId snapshotId;
} PTS_SnapshotSync_Request;

-Generic Response-
```

Description

This function synchronizes the snapshot identified by *snapshotId* with a TPM PCR by calling TPM_Extend calls. A snapshot containing a history of extended values called a synchronization snapshot may be created.

This function is invoked by the PTS client.

Input Parameters	Description
snapshotId	Identifies a snapshot to be synchronized. The snapshot is extended into a TPM PCR

Error Codes	Condition
PTS_INSUFFICIENT_RESOURCES	If creating a new sync snapshot – may encounter
PTS_FILE_SYSTEM_ERROR	If creating a new sync snapshot – may encounter
PTS_SNAPSHOT_NOT_FOUND	Snapshot ID does not exist

6.4.6 PTS_SnapshotSyncComplete

```
typedef struct {
      PTS_SnapshotId          snapshotId;
      PTS_SnapshotId          syncId;
} PTS_SnapshotSyncComplete_Async_Response;
```

Description

This command is originated by the PTS in response to the client call to PTS_SnapshotSync.

This function confirms the synchronization of the snapshot identified by *snapshotId*. PTS maintains an internal snapshot structure containing a history of extended values called a synch-snapshot. The sync-snapshot is identified by *syncId*.

This function is invoked by the PTS.

Input Parameters	Description

snapshotId	Identifies the snapshot synchronized to a TPM PCR.
syncId	Identifies the snapshot consisting of hash results of other snapshots where these values have been extended into a TPM PCR.

Error Codes	Condition
PTS_INSUFFICIENT_RESOURCES	If creating a new sync snapshot – may encounter
PTS_FILE_SYSTEM_ERROR	If creating a new sync snapshot – may encounter
PTS_SNAPSHOT_NOT_FOUND	Snapshot ID does not exist
PTS_SYNC_SNAPSHOT_NOT_FOUND	If PTS doesn't provide access control to sync snapshot + lots of other ways to delete it

6.4.7 PTS_SnapshotVerify

```
typedef struct {
      PTS_SnapshotId    snapshot;
      PTS_UInt32        verifyDepth;
      PTS_Bool          verboseOutputFlag;
      PTS_UInt32        numRules;
      PTS_UUID          rules[];
} PTS_SnapshotVerify_Request;

typedef struct {
      PTS_SnapshotId result;
} PTS_SnapshotVerify_Response;
```

Description

This command is implemented by the PTS and is called by the client process.

The snapshot structure is verified according to the rules specified in rules.

A new snapshot is generated by PTS. The resultant snapshotId is returned in the response message. If verboseOutputFlag is TRUE, then in addition to the verification results the result snapshot also includes all values and assertions contained in the input snapshot.

The purpose of this command is to verify a snapshot's contents against a rule so that the result of the verify process can be reported rather than the full detail of the original snapshot if verboseOutputFlag is FALSE.

Input Parameters	Description
Snapshot	Identifies the layer 0 snapshots to be verified. If verifyDepth is >0 then lower layer snapshot structures will be verified accordingly.
verifyDepth	Specifies the number of layers of sub-components to be traversed. A verifyDepth of MAXINT will traverse to component leaf nodes.
verboseOutputFlag	If TRUE, the result snapshot includes all measurement values and assertions from

	contained in the input snapshot in addition to the verification results populated in AssertionInfo. If FALSE, the result snapshot only contains the verification results, but no measurement values or assertions from the input snapshot.
numRules	The number of rule identifiers in *rules*
Rules	Identifies particular reference measurements / rules to be applied when computing measurements and verifying them. (e.g. RIMM.xs:ID). If numRules is 0, a best effort attempt will be made to locate an applicable RIMM / rule given ComponentId.

Output Parameters	**Description**
Result	Snapshot ID of the snapshot containing the results of the verification actions.

Error Codes	**Condition**
PTS_SNAPSHOT_NOT_FOUND	A snapshot corresponding to the supplied snapshotId could not be found
PTS_VERIFY_FAILED	If the measured values differ from reference measurements.
PTS_RULE_NOT_FOUND	The rules could not be found.
PTS_INVALID_RULE	Can't parse or process rule
PTS_RULE_NOT_AUTHORIZED	Rule password protected or password not accepted

6.4.8 PTS_ReportVerify

```
typedef struct {
      PTS_Bool               verboseOutputFlag;
      PTS_UInt32             numRules;
      PTS_UUID               rules[];
      PTS_IntegrityReport    report;
} PTS_ReportVerify_Request;
```

```
typedef struct {
      PTS_SnapshotId         result;
      PTS_Handle             handle;
} PTS_ReportVerify_Response;
```

Description

This command is implemented by the PTS and is called by the client process.

The report structure is verified according to the rules specified in rules.

A new snapshot is generated by PTS. The resultant snapshotId is returned in the response message. If verboseOutputFlag is TRUE, the snapshot containing the verification results will be

appended to the existing integrity report and assigned a new report identier returned as handle. If verboseOutputFlag is FALSE, then handle is nil.

Input Parameters	Description
verboseOutputFlag	If TRUE, the snapshot containing the verification results is appended to the integrity report identified by report. A handle to the resulting integrity report is returned in handle. If FALSE, the returned handle is nil.
numRules	The number of rule identifiers in *rules*
rules	Identifies a particular reference measurement / rule to be applied when computing measurements and verifying them. (e.g. RIMM.xs:ID). If numRules is 0, a best effort attempt will be made to locate an applicable RIMM / rule given ComponentId.
report	Identifies the integrity report to be verified.

Output Parameters	Description
result	Snapshot Id of the snapshot containing the results of the verfication actions.
handle	If verboseOutputFlag is TRUE, handle references a new report that contains all of the data of the original report and the new snapshot containing the verification results. Handle is nil if verboseOutputFlag is FALSE.

Error Codes	Condition
PTS_VERIFY_FAILED	If the measured values differ from reference measurements.
PTS_RULE_NOT_FOUND	The rules could not be found.
PTS_INVALID_RULE	Can't parse or process rule
PTS_RULE_NOT_AUTHORIZED	Rule password protected or password not accepted

6.5 Snapshot Creation Commands

Integrity logging and reporting functions render integrity values in a vendor neutral interoperable format. The format used is specified by the IWG Core Integrity Schema [6].

All functions in this section are implemented by the PTS and called by a local process.

6.5.1 PTS_SnapshotCreate

```
typedef struct {
      PTS_UUID    ownerId;
} PTS_SnapshotCreate_Request;

typedef struct {
```

```
      PTS_SnapshotId newSnapshotId;
} PTS_SnapshotCreate_Response;
```

Description

This function creates a snapshot resource and returns a unique identifier.

Input Parameters	Description
ownerId	The snapshot owner other than the creator. If ownerId is Nil, i.e. the UUID 00000000-0000-0000-0000-000000000000 (see RFC 4122 section 4.1.7), then the session clientId is used as the snapshot ownerId. PTS associates ownerId persistently with the snapshot until the snapshot is deleted.

Output Parameters	Description
newSnapshotId	The snapshot identifier is generated by PTS. The identifier is a globally unique UUID.

6.5.2 PTS_SnapshotDelete

```
typedef struct {
      PTS_SnapshotId snapshotId
} PTS_SnapshotDelete_Request;

-Generic Response-
```

Description

This function deletes the snapshot record from the PTS integrity log.

Input Parameters	Description
snapshotId	A snapshot identifier

Error Codes	Condition
PTS_SNAPSHOT_NOT_FOUND	The identified snapshot cannot be found.
PTS_DENIED	The identified snapshot may not be deleted.

6.5.3 PTS_SnapshotImport

```
typedef struct {
      PTS_UUID    ownerId;
      PTS_UInt32  size;
      PTS_Byte    snapshotXml;
} PTS_SnapshotImport_Request;

typedef struct {
      PTS_SnapshotId newSnapshotId;
} PTS_SnapshotImport_Response;
```

Description

This function imports a snapshot in XML format and returns the snapshot identifier.

The snapshot identifier is taken from the snapshot xs:ID attribute. If there is another snapshot with the same ID already in PTS an error is returned.

Input Parameters	Description
ownerId	The snapshot owner other than the creator. If ownerId is Nil i.e. the UUID 00000000-0000-0000-0000-000000000000 (see RFC 4122, section 4.1.7), then the session clientId is used as the snapshot ownerId. PTS associates ownerId persistently with the snapshot until the snapshot is deleted.
size	Size in bytes of snapshotXML
snapshotXml	The snapshot rendered in XML format

Output Parameters	Description
newSnapshotId	The snapshot identifier is generated by PTS. The identifier is a globally unique UUID.

Error Codes	Condition
PTS_DUPLICATE_SNAPSHOT	A snapshot with the same Id already exists in PTS
PTS_UUID_REUSED	The supplied UUID has already been used to uniquely identify another (non-clientId) PTS structure
PTS_INVALID_XML	The supplied XML snapshot is invalid

6.5.4 PTS_SnapshotExport

```
typedef struct {
      PTS_SnapshotId snapshotId;
} PTS_SnapshotImport_Request;

typedef struct {
      PTS_UInt32  size;
      PTS_Byte    snapshotXml;
} PTS_SnapshotImport_Response;
```

Description

This function exports a copy of the snapshot identified by snapshotId in XML format. An original copy is retained in PTS.

Input Parameters	Description
snapshotId	The snapshot identifier is used to lookup the snapshot.

Output Parameters	Description
size	Size in bytes of snapshotXML
snapshotXml	The snapshot rendered in XML format.

Error Codes	Condition
PTS_SNAPSHOT_NOT_FOUND	The identified snapshot cannot be found

6.5.5 PTS_SnapshotGetProperties

```
typedef struct {
      PTS_SnapshotId snapshotId;
} PTS_SnapshotGetProperties_Request;

typedef struct {
      PTS_SnapshotProperties  properties;
} PTS_SnapshotGetProperties_Response;
```

Description

This command returns the properties of a snapshot.

Input Parameters	Description
snapshotId	A snapshot identifier

Output Parameters	Description
properties	Temporal properties of the snapshot

Error Codes	Condition
PTS_SNAPSHOT_NOT_FOUND	The identified snapshot cannot be found

6.5.6 PTS_SnapshotOpen

```
typedef struct {
      PTS_SnapshotFlags flags;
      PTS_SnapshotId snapshotId;
} PTS_SnapshotOpen_Request;

typedef struct {
      PTS_SnapshotDescriptor  snapshotDescriptor;
} PTS_SnapshotOpen_Response;
```

Description

This command prepares as snapshot for update. The PTS allocates a descriptor resource that is meaningful in the context of the current session and snapshot manipulation commands. Descriptors are automatically recycled when the session ends.

Input Parameters	Description
snapshotFlags	Specifies the mode of access and sharing
snapshotId	A snapshot identifier

Output Parameters	Description
snapshotDescriptor	Descriptor for accessing a snapshot.

Error Codes	Condition
PTS_SNAPSHOT_NOT_FOUND	Snapshot ID does not exist
PTS_INVALID_FLAGS	Illegal or illogical flag combination.
PTS_SNAPSHOT_ACCESS_DENIED	Snapshot not available for requested mode of access and sharing.

6.5.7 PTS_SnapshotClose

```
typedef struct {
      PTS_snapshotDescriptor;
} PTS_SnapshotClose_Request;

-Generic Response-
```

Description

This command closes the snapshot for update.

Input Parameters	Description
snapshotDescriptor	Descriptor for accessing a snapshot.

Error Codes	Condition
PTS_INVALID_DESCRIPTOR	The descriptor does not refer to a valid snapshot

6.5.8 PTS_SnapshotUpdateComponentId

```
typedef struct {
      PTS_SnapshotDescriptor  snapshotDescriptor;
      PTS_ComponentId         newComponentId;
} PTS_SnapshotUpdateComponentId_Request;

-Generic Response-
```

Description

This function writes values to the ComponentId element in the snapshot.

Input Parameters	Description
snapshotDescriptor	Descriptor for writing to a snapshot structure
newComponentId	ComponentId data that replaces any previous ComponentId values.

Error Codes	Condition
PTS_INVALID_PARAMETER	The ComponentId structure in invalid
PTS_INVALID_DESCRIPTOR	The descriptor does not refer to a valid snapshot

6.5.9 PTS_SnapshotUpdateSubComponents

```
#define PTS_SN_OPCODE_ADD     0
#define PTS_SN_OPCODE_DEL     1

typedef struct {
      PTS_SnapshotDescriptor        snapshotDescriptor;
      PTS_UInt32                    opcode;
      PTS_VariableLengthDataPtr     cid; //PTS_ComponentId
      PTS_VariableLengthDataPtr     uri; //PTS_String
      PTS_VariableLengthDataArea    Data;
} PTS_SnapshotUpdateSubComponents_Request;

-Generic Response-
```

Description

This function adds or removes entries in the snapshot's list of subcomponents.

Input Parameters	Description
snapshotDescriptor	Descriptor for writing to a snapshot structure
opcode	If operation is PTS_SN_OPCODE_ADD then cid is added to the list of snapshot subcomponents. If PTS_SN_OPCODE_DEL then the matching entry is deleted.
cid	Component ID for a sub-component of this snapshot.
uri	If uri is non-null and operation is PTS_SN_OPCODE_ADD then the location of the subcomponent – its URI - is included in the subcomponent entry.

Error Codes	Condition
PTS_INVALID_OPCODE	The opcode specified does not support the input parameters
PTS_OPCODE_NOT_DEFINED	The specified opcode is not defined

PTS_INVALID_PARAMETER	The ComponentId structure in invalid
PTS_INVALID_DESCRIPTOR	The descriptor does not refer to a valid snapshot
PTS_COMPONENT_REF_EXISTS	The specified component ID already exists. The previous entry remains unchanged. May be returned in response to PTS_SN_OPCODE_ADD.
PTS_COMPONENT_NOT_FOUND	The specified component ID does not exist. May be returned in response to PTS_SN_OPCODE_DEL.

6.5.10 PTS_SnapshotUpdateAssertions

```
typedef struct {
      PTS_SnapshotDescriptor  snapshotDescriptor;
      PTS_AssertionInfo       newAssertions;
} PTS_SnapshotUpdateAssertions_Request;

-Generic Response-
```

Description

This function includes the XML assertions into the Assertions element in the snapshot. The PTS may incorporate these assertions through schema extension. The assertions are formatted XML that PTS may parse to locate schemas for constructing the appropriate XML based on extension of TCG schemas for reporting assertions.

Input Parameters	Description
snapshotDescriptor	Descriptor for writing to a snapshot structure
newAssertions	A list of assertions that replaces any previous list of assertions. Assertions are expressed in XML

Error Codes	Condition
PTS_INVALID_DESCRIPTOR	The descriptor does not refer to a valid snapshot
PTS_INVALID_PARAMETER	The assertions are not expressed in XML

6.5.11 PTS_SnapshotUpdateIntegrityValues

```
typedef struct {
      PTS_SnapshotDescriptor  snapshotDescriptor;
      PTS_UInt32              numValues;
      PTS_ValuesInfo          newValues[];
} PTS_SnapshotUpdateIntegrityValues_Request;

-Generic Response-
```

Description

This function writes integrity measurement values into the snapshot.

Input Parameters	Description
snapshotDescriptor	Descriptor for writing to a snapshot structure

numValues	The number of newValues in the array.
newValues	A list of integrity measurement values that replaces any previous list of values. Values are expressed as message digests or optionally as opaque (I.e. not interpreted) structures.

Error Codes	Condition
PTS_INVALID_DESCRIPTOR	The descriptor does not refer to a valid snapshot
PTS_INVALID_PARAMETER	One or more of the PTS_ValuesInfo parameters is invalid

6.5.12 PTS_SnapshotUpdateIntegrityValuesXml

```
typedef struct {
      PTS_SnapshotDescriptor  snapshotDescriptor;
      PTS_UInt32              numValues;
      PTS_String              newXmlValues[];
} PTS_SnapshotUpdateIntegrityValuesXml_Request;

-Generic Response-
```

Description

This function writes XML integrity measurement values into the snapshot.

Input Parameters	Description
snapshotDescriptor	Descriptor for writing to a snapshot structure
numValues	The number of newValues in the array.
newXmlValues	A list of XML integrity measurement values that replaces any previous list of values.

Error Codes	Condition
PTS_INVALID_DESCRIPTOR	The descriptor does not refer to a valid snapshot
PTS_INVALID_PARAMETER	One or more of the PTS_ValuesInfo parameters is invalid

6.5.13 PTS_SnapshotUpdateCollector

```
typedef struct {
      PTS_SnapshotDescriptor  snapshotDescriptor;
      PTS_ComponentId         newCollector;
} PTS_SnapshotUpdateCollector_Request;

-Generic Response-
```

Description

This function writes the collector ComponentId into the snapshot's Collector element.

Input Parameters	Description
snapshotDescriptor	Descriptor for writing to a snapshot structure
newCollector	A ComponentId of the collector

Error Codes	Condition
PTS_INVALID_DESCRIPTOR	The descriptor does not refer to a valid snapshot
PTS_INVALID_PARAMETER	One or more of the Component ID paramters is invalid

6.6 Reporting Commands

6.6.1 PTS_ReportCreate

```
- Generic Request -

typedef struct {
      PTS_Handle        reportHandle;
} PTS_ReportCreate_Response;
```

Description

This function creates a context within PTS for an integrity report.

Output Parameters	Description
reportHandle	A handle used to reference an integrity report context.

Error Codes	Condition
PTS_INVALID_HANDLE	Invalid handle
PTS_HANDLE_EXIST	Attempt to create context using an existing handle

6.6.2 PTS_ReportDelete

```
typedef struct {
      PTS_Handle        reportHandle;
} PTS_ReportDelete_Request;

- generic response –
```

Description

This function deletes the integrity report context identified by reportHandle.

Input Parameters	Description
reportHandle	A handle used to reference an integrity report context.

Error Codes	Condition
PTS_REPORT_NOT_FOUND	The specified reportHandle does not reference a valid integrity report.
PTS_DENIED	The context idenfied by the handle may not be deleted

6.6.3 PTS_ReportSpecify

```
typedef struct (
      PTS_Handle                handle;
      PTS_UInt32                flags;
      PTS_UInt32                numComponents;
      PTS_VariableLengthDataPtr componentSelectors[];
                                          // PTS_AddByComponent
      PTS_AddByTrustChain       chainSelector;
      PTS_VariableLengthDataPtr collectorSelector; //PTS_AddByCollector
      PTS_AddByOwner            ownerSelector;
      PTS_VariableLengthDataPtr pcrSelector;
      PTS_VariableLengthDataArea  data;
} PTS_ReportSpecify_Request;

- generic response –
```

Description

An integrity report is populated according to the constraints specified. Boolean flags are used to control the scope of the report. Snapshots are included according to various criteria:

- Component – where the component ID of the subsystem of interest is known
- Component subtree – where the subsystem of interest is comprised of subordinate components
- Trust Chain – where a component is part of a transitive trust chain
- Collector – where the component that is performing measurement collection is specified in the snapshot
- Owner – for a report concerning the entity that created snapshots

By specifying the same componentId for each dimension, the intersection of criteria can be achieved.

By specifying the partialMatch flag in PTS_AddByComponent a componentId, (e.g. where only VendorId is specified), can be used to select a range of snapshots that have a common dimension for the report.

Input Parameters	Description
handle	The handle to a report context.
flags	IF bit-0 = TRUE, componentSelectors is evaluated. IF bit-1 = TRUE, chainSelector is evaluated. IF bit-2 = TRUE, collectorSelector is evaluated. IF bit-3 = TRUE, ownerSelector is evaluated. IF bit-4 = TRUE, pcrSelector is evaluated.
numComponents	The number of PTS_AddByComponent structures.

componentSelectors	Selects snapshots to be included in an integrity report based on the snapshot ComponentId.
chainSelector	Selects snapshots to be included in an integrity report based on Transitive Trust Chain.
collectorSelector	Selects snapshots to be included in an integrity report based on the component that performed the measurement and collection operations.
ownerSelector	Selects snapshots to be included in an integrity report based on the snapshot owner. Owner = ClientId
pcrSelector	Selects snapshots to be included in an integrity report if they correspond to the PCRs indicated in *pcrSelctor*

Error Codes	**Condition**
PTS_INVALID_HANDLE	Supplied handle does not match internal context
PTS_INVALID_PARAMETER	A input parameter is malformed or wrong number of components
PTS_SNAPSHOT_NOT_FOUND	The specified component selector does not match any snapshot
PTS_INVALID_TTCHAIN	The transitive trust chain does not exist or is malformed
PTS_INVALID_COLLECTOR	The specified collector is invalid.
PTS_UNKNOWN_OWNER	The no such owner.
PTS_INVALID_PCR_SELECTION	The PCR selection does not correspond to physical PCRs

6.6.4 PTS_ReportGenerate

```
typedef struct (
      PTS_Handle                handle;
      PTS_UInt32                flags;
      PTS_VariableLengthDataPtr pcrs; // PTS_PcrBitmask
      PTS_UInt64                pdpNonce;
      PTS_VariableLengthDataPtr canonicalizationAlg; // PTS_AlgorithmId
      PTS_UInt32                confidenceValue;
      PTS_UInt32                confidenceBase;
      PTS_VariableLengthDataPtr signerInfo; // PTS_SignerInfo
      PTS_Key                   quoteKey;
                                PTS_VariableLengthDataArea   data;
} PTS_ReportGenerate_Request;

typedef struct {
      PTS_IntegrityReport xmlReport;
} PTS_ReportGenerate_Response;
```

Description

This function renders the integrity report corresponding to that specified by PTS_ReportSpecify and by pcrs in PTS_ReportGenerate in XML format.

The component signing the report is PTS. The PTS componentId is used to populate the SigningComponent element in the XML schema.

Input Parameters	Description
handle	Report handle.
flags	Bit 0: If TRUE, a TPM Quote operation will be performed using the PCR selection in *pcrs*. Bit 1: If TRUE, a TPM Sign operation will be performed.
pcrs	A list of PCR numbers associated with the TPM. IF pcrs.sizeOfSelect = 0, THEN all snapshots corresponding to PCR values in pcrs are included in the report. If FALSE the pcrs parameter is ignored.
pdpNonce	A nonce supplied by the entity receiving this integrity report. The same nonce is used for both the Quote and the Signature operations. The current TPM specification requires 20-byte nonces. The PTS implementer is responsible for transforming pdpNonce into the appropriate length to support TPM structures. IF pdpNonce is supplied then the report SHOULD be signed. I.e. flags.Bit-1 SHOULD be TRUE.
signerInfo	The key used to sign the report. PTS uses the signKey to obtain signing algorithm, key size and other attributes in the report. IF flags.Bit-1 is FALSE and flags.Bit-0 is TRUE, THEN signerInfo.signingKey value is ignored.
quoteKey	The key used to perform a TPM_Quote operation.

Output Parameters	Description
xmlReport	All snapshot structures that have been measured are returned as an XML formatted structure.

Error Codes	Condition
PTS_INVALID_HANDLE	The report handle is invalid
PTS_TPM_NOT_FOUND	If no TPM is installed and doQuote is specified, a TPM NOT FOUND error is returned.
PTS_INVALID_TPM_LOG	The TPM log is unavailable or cannot be converted to a snapshot structure
PTS_QUOTE_FAILED	Quote operation failed
PTS_ALREADY_SIGNED	The object to be signed already contains a

	signature.
PTS_SIGN_FAILED	Signing operation failed
PTS_KEY_NOT_FOUND	The desired key could not be found
PTS_TPM_OTHER_ERROR	TPM generic failure
PTS_INVALID_CANONICALIZATION	The specified canonicalization function is malformed or cannot be applied.
PTS_INVALID_CONFIDENCE	The confidenceBase is zero

6.6.5 PTS_ReportGetProperties

```
typedef struct {
      PTS_Handle        reportHandle;
} PTS_ReportDelete_Request;

typedef struct {
      PTS_reportProperties;
} PTS_ReportGetProperties_Response;
```

Description

This command returns the properties of a report.

Input Parameters	Description
reportHandle	A handle used to reference an integrity report context.

Output Parameters	Description
reportProperties	Temporal properties of the integrity report

Error Codes	Condition
PTS_REPORT_NOT_FOUND	The specified reportHandle does not reference a valid integrity report.

6.6.6 PTS_SnapshotSign

```
typedef struct {
      PTS_SnapshotId    snapshotId;
      PTS_UInt64        pdpNonce;
      PTS_Bool          forceSign;
      PTS_SignerInfo    signerInfo;
} PTS_SnapshotSign_Request;

-Generic response-
```

Description

This function digitally signs a snapshot.

The component signing the report is PTS. The PTS componentId is used to populate the SigningComponent element in the XML schema.

PTS_SnapshotSign does not provide a means to specify a Transform Method. PTS will utilize its default transform method. PTS implementers are free to provide a means to configure the default signing transform method at installation time or via a vendor-specific extension.

Input Parameters	Description
`snapshotId`	Identifies a snapshot
`pdpNonce`	A nonce supplied by the entity receiving this integrity report. The current TPM specification requires 20-byte nonces. The PTS implementer is responsible for transforming pdpNonce into the appropriate length to support TPM structures.
`forceSign`	Normally the signature operation will fail if the snapshot is already signed. If forceSign flag is TRUE, the existing signature is replaced with a new one.
`signerInfo`	The key used to sign the snapshot. PTS uses the signingKey to obtain signing algorithm, key size and other attributes necessary to populate the signature elements in the snapshot. Specifies the algorithm used to remove whitespace prior to signing XML structures. Signer's confidence level that collected measurements are accurate Divisor applied to confidenceValue

Error Codes	Condition
`PTS_INVALID_SNAPSHOT`	The specified snapshot is malformed
`PTS_SNAPSHOT_NOT_FOUND`	The specified snapshot cannot be located
`PTS_ALREADY_SIGNED`	The object to be signed already contains a signature.
`PTS_SIGN_FAILED`	Signing operation failed
`PTS_KEY_NOT_FOUND`	The desired signing key could not be found
`PTS_INVALID_CANONICALIZATION`	The specified canonicalization function is malformed or cannot be applied.
`PTS_INVALID_CONFIDENCE`	The confidenceBase is zero

6.7 PTS Configuration Commands

6.7.1 PTS_RegisterRule

```
#define ENCODING_UNDEFINED 0
#define ENCODING_XML 1
```

```
typedef struct {
      PTS_Key     unsealKey;
      PTS_VariableLengthDataPtr    passPhrase; // PTS_String passPhrase
      PTS_UInt32  ruleEncoding;
      PTS_VariableLengthDataPtr    rule; // PTS_Byte ruleXML
      PTS_VariableLengthDataArea   data;
} PTS_RegisterRule_Request;

typedef struct {
      PTS_UUID ruleId;
} PTS_RegisterRule_Response;
```

Description

This command is implemented by the PTS and is called by a management process.

The caller provisions PTS with a collection of rules that can be used to validate scan results.

If the platform contains a TPM, it is possible that rules have been encrypted using a TPM seal or bind operation. If sealed a TPM_Unseal / TPM_Unbind must be used to obtain cleartext representation of the rules. Sealing can ensure that PTS does not perform checks while in an untrustworthy state.

Rules are expressions that can be used by PTS to verify collected measurements. Typically, rules are expressed in XML (hence, they are self-describing). TCG Reference Integrity Measurement Manifest (RIMM) structures may be supplied as rules. A RIMM rule instructs PTS to verify a snapshot using the RIMM for reference measurements.

More sophisticated rules expressions may be supplied instructing PTS to perform more sophisticated verification actions. It is anticipated that PTS extensibility features may be needed to fully apply the dictates of the rule.

It is assumed the unseal / unbind key was created by another process / application.

Input Parameters	**Description**
`unsealKey`	Identifies the key used to unseal/unbind rules for PTS use. If unsealKey.keyLength is 0, then rules are in cleartext.
`passPhraseSize`	Length (in bytes) of pass phrase string
`passPhrase`	Passphrase string
`ruleEncoding`	A tag indicating the format in which rules.rule is encoded.
`Rule`	An XML rule expression. The XML rule MUST contain a URI.
`Data`	A buffer containing the pass phrase and the rule

Output Parameters	Description
`ruleId`	A UUID for the registered rule. (The UUID is typically obtained from the UUID or xs:ID of an XML expression.) If the rule is a Reference Manifest, the ruleID MUST be the Reference Manifest UUID.

Error Codes	Condition
PTS_KEY_NOT_FOUND	Unable to find unseal key
PTS_KEY_LENGTH_UNSUPPORTED	Unseal key length not supported
PTS_CORRUPT_KEY	Key blob is corrupt
PTS_INVALID_PASS_PHRASE	Incorrect unseal key pass phrase
PTS_INVALID_XML	Invalid rule encoding, rule not in XML, or rule XML invalid
PTS_INSUFFICIENT_RESOURCES	Not enough memory or persistent storage; or data structure limitation reached.

6.7.2 PTS_UnregisterRule

```
 typedef struct {
      PTS_UUID    ruleId;
} PTS_UnregisterRule_Request;

- generic response -
```

Description

This command is implemented by the PTS and is called by a management process.

The rule specified by ruleId is removed from the working set of rules.

Input Parameters	Description
ruleId	A UUID for the registered rule.

Error Codes	Condition
PTS_RULE_NOT_FOUND	Rule not found

6.7.3 PTS_ListRules

```
- generic request -

typedef struct {
      PTS_UInt32        size;
      PTS_Bytes         rulesXML; // rules rendered in XML
} PTS_ListRules_Response;
```

Description

This command is implemented by the PTS and is called by a management process.

The set of all rules currently registered with PTS is output rendered in XML format.

Output Parameters	Description

size	Length (in bytes) of rulesXML
rulesXML	The currently registered rules rendered in XML format.

Error Codes	Condition

6.7.4 PTS_ConfigurePCR

```
typedef struct {
      PTS_PcrId   Pcr;
} PTS_ConfigurePCR_Request;


-Generic Response-
```

Description:

This command is implemented by the PTS and is called by a management process.

The PTS_ConfigurePCR command is used to choose the PCR that PTS may use.

Input Parameters	Description
Pcr	The PCR number that PTS may use to extend snapshot data

Error Codes	Condition
PTS_INVALID_PCR_SELECTION	Specified PCR already reserved or does not exist

6.7.5 PTS_RegisterQuoteKey

```
typedef struct {
      PTS_Key     quoteKey;
      PTS_UInt16  passPhraseSize;
      PTS_Byte    passPhrase;       // First byte of pass phrase
      PTS_Key     storeKey;         // storage key
      PTS_UInt32  authSizeSk;
      PTS_Byte    authDataSk;       // storage auth value
} PTS_RegisterQuoteKey_Request;

-Generic Response-
```

Description

This command is implemented by the PTS and is called by a management process.

The caller provisions PTS with a UUID of a key that will be used for quoting TPM PCRs.

authDataSk is used to access a protected storage system or to access a storage key identified by storeKey which can be used for integrity protecting the quoteKey. PTS should use the TPM key storage or TPM Non-Volatile storage capabilities if a TPM is available on the platform.

Input Parameters	Description
quoteKey.keyLength	Length (in bytes) of the key
quoteKey.keyId	UUID of the signing key
passPhraseSize	Length (in bytes) of the pass phrase string
passPhrase	Pass phrase string
storeKey.keyLength	Length (in bytes) of the key
storeKey.keyId	Identifier of the storage key. Contents of which is implementation specific. Normally this is a UUID.
authSizeSk	Length (in bytes) of authorization data
authDataSk	Authorization data for accessing storage system

Error Codes	Condition
PTS_KEY_NOT_FOUND	Unable to find quote key
PTS_KEY_LENGTH_UNSUPPORTED	Quote key length not supported
PTS_CORRUPT_KEY	Key object is corrupted
PTS_INVALID_PASS_PHRASE	Incorrect quote key pass phrase
PTS_AUTHENTICATION_FAILURE	Unable to authenticate to key storage device
PTS_INSUFFICIENT_RESOURCES	Not enough memory or persistent storage; or data structure limitation reached.

6.7.6 PTS_UnregisterQuoteKey

```
typedef struct {
      PTS_Key     quoteKey;
      PTS_Key     storeKey;         // storage key
      PTS_UInt32  authSizeSk;
      PTS_Byte    authDataSk;       // storage auth value
} PTS_UnregisterQuoteKey_Request;

-Generic Response-
```

Description

This command is implemented by the PTS and is called by a management process.

The caller specifies a key to be removed from the PTS list of registered quote keys.

Input Parameters	Description
quoteKey.keyLength	Length (in bytes) of the key
quoteKey.keyId	UUID of the signing key
storeKey.keyLength	Length (in bytes) of the key
storeKey.keyId	Identifier of the storage key. Contents of which is implementation specific. Normally this is a UUID.

authSizeSk	Length (in bytes) of authorization data
authDataSk	Authorization data for accessing storage system

Error Codes	Condition
PTS_KEY_NOT_FOUND	Unable to find quote key
PTS_KEY_LENGTH_UNSUPPORTED	Quote key length not supported
PTS_CORRUPT_KEY	Key object is corrupted
PTS_AUTHENTICATION_FAILURE	Unable to authenticate to key storage device

6.7.7 PTS_ListQuoteKeys

```
- generic request -

typedef struct {
      PTS_UInt32        numKeys;
      PTS_Key           quoteKeys; // list of registerd keys
} PTS_ListQuoteKeys_Response;
```

Description

This command is implemented by the PTS and is called by a client.

The set of currently registered quote keys is returned.

Output Parameters	Description
numKeys	The number of quoteKeys
quoteKeys	The currently registered list of quote keys

Error Codes	Condition
PTS_CORRUPT_KEY	A key object is corrupted

6.7.8 PTS_RegisterSigningKey

```
typedef struct {
      PTS_Key     signingKey;
      PTS_UInt16  passPhraseSize;
      PTS_Byte    passPhrase;       // First byte of pass phrase
      PTS_Key     storeKey;         // storage key
      PTS_UInt32  authSizeSk;
      PTS_Byte    authDataSk;       // storage auth value
} PTS_RegisterSigningKey_Request;

-Generic Response-
```

Description

This command is implemented by the PTS and is called by a management process.

The caller provisions PTS with an ID of a key that will be used by the PTS to sign snapshot structures that are created and managed by the PTS (e.g. the “sync” snapshot).

authDataSk is used to access a protected storage system or to access a storage key identified by storeKey which can be used for integrity protecting the signingKey. PTS should use the TPM key storage or TPM Non-Volatile storage capabilities if a TPM is available on the platform.

Input Parameters	Description
Key.keyLength	Length (in bytes) of the key
Key.keyId	Identifier of the signing key. Contents of which is implementation specific. Normally this is a UUID.
passPhraseSize	Length (in bytes) of pass phrase string
passPhrase	Pass phrase string
storeKey.keyLength	Length (in bytes) of the key
storeKey.keyId	Identifier of the storage key. Contents of which is implementation specific. Normally this is a UUID.
authSizeSk	Length (in bytes) of authorization data
authDataSk	Authorization data for accessing storage system

Error Codes	Condition
PTS_KEY_NOT_FOUND	Unable to find signing key
PTS_KEY_LENGTH_UNSUPPORTED	Singing key length not supported
PTS_CORRUPT_KEY	Key blob is corrupt
PTS_INVALID_PASS_PHRASE	Incorrect signing key pass phrase
PTS_AUTHENTICATION_FAILURE	Unable to authenticate to key storage device
PTS_INSUFFICIENT_RESOURCES	Not enough memory or persistent storage; or data structure limitation reached.

6.7.9 PTS_UnregisterSigningKey

```
typedef struct {
      PTS_Key     quoteKey;
      PTS_Key     storeKey;         // storage key
      PTS_UInt32  authSizeSk;
      PTS_Byte    authDataSk;       // storage auth value
} PTS_UnregisterSigningKey_Request;

-Generic Response-
```

Description

This command is implemented by the PTS and is called by a management process.

The caller specifies a key to be removed from the PTS list of registered signing keys.

Input Parameters	Description

signingKey.keyLength	Length (in bytes) of the key
signingKey.keyId	UUID of the signing key
storeKey.keyLength	Length (in bytes) of the key
storeKey.keyId	Identifier of the storage key. Contents of which is implementation specific. Normally this is a UUID.
authSizeSk	Length (in bytes) of authorization data
authDataSk	Authorization data for accessing storage system

Error Codes	Condition
PTS_KEY_NOT_FOUND	Unable to find key
PTS_KEY_LENGTH_UNSUPPORTED	Key length not supported
PTS_CORRUPT_KEY	Key object is corrupted
PTS_AUTHENTICATION_FAILURE	Unable to authenticate to key storage device

6.7.10 PTS_ListSigningKeys

```
- generic request -

typedef struct {
      PTS_UInt32        numKeys;
      PTS_Key           signingKeys; // list of registerd keys
} PTS_ListSigningKeys_Response;
```

Description

This command is implemented by the PTS and is called by a client.

The set of currently registered signing keys is returned.

Output Parameters	Description
numKeys	The number of signingKeys
signingKeys	The currently registered list of signing keys

Error Codes	Condition
PTS_CORRUPT_KEY	A key object is corrupted

6.7.11 PTS_GetCapabilities

```
typedef struct {
      PTS_UInt32  vendorId;
      PTS_UInt32  commandOrdinal;
} PTS_GetCapabilities_Request;

typedef struct {
```

```
      PTS_VariableLengthDataArea capabilityList[]; // an array of
                                                    PTS_Capability
} PTS_GetCapabilities_Response;
```

Description

This function is implemented by the PTS and called by client processes seeking to query the PTS on supported functionality.

The caller identifies the vendorId and the command ordinal for which implementation status information is required.

The vendorId `0x00000000` identifies commands specified in this specification.

The vendorId `0xffffffff` indicates all vendorIds.

The command ordinal `0xffffffff` is used to indicate all commands.

A request with the command ordinal `0xffffffff` SHOULD return a list of all commands supported for the indicated vendorId. A request with the vendorId `0xffffffff` and the commandOrdinal `0xffffffff` SHOULD return a list of all commands supported by the PTS implementation.

The implementationStatus field specifies whether capabilities information exists for the command. A value of zero indicates that the command identified in the request has not been implemented. A value of one indicates that the command has been implemented. An implementation may indicate partial implementation of the command by setting bit 0 to one and setting appropriate bits in the mask to zero or one. A feature mask is not defined for all commands.

A vendor may indicate lack of support for the PTS_GetCapabilities command by returning the status result **PTS_COMMAND_NOT_IMPLEMENTED.**

Name	Description
`PTS_ComponentScan`	bit 1: in memory scan support bit 2: persistent storage (file scan) support bit 3: verification support bit 4: interval scanning support bit 5: depth support
`PTS_ReportSpecify`	bit 1: addByComponent supported bit 2: addByComponent partial match supported bit 3: addByTrustchain supported bit 4: addByCollector supported bit 5: addByOwner supported bit 6: addByPCR supported
`PTS_GetCapabilities`	bit 1: Persistent snapshots bit 2: Persistent reports bit 3: TPM utilized bit 4: Report Generation

PTS_ReportGenerate	bit 1: sign support bit 2: TPM quote support
PTS_RegisterRule	bit 1: RIMM support bit 2: non-RIMM rule support bit 3: seal support bit 4: pass phrase support
PTS_ConfigureQuoteKey	bit 1: pass phrase support
PTS_ConfigureSigningKey	bit 1: pass phrase support

Input Parameters	Description
vendorId	SMI private enterprise number indicates a vendor specific extension. Zero (0x0) indicates command ordinals defined by this specification. This is the PTS VendorId – for vendor-specific commands. The value 0x00000000 is used to indicate IF-PTS commands defined in this specification. The value 0xffffffff is used to indicate all vendorIds.
commandOrdinal	Ordinal of the command being queried. If the ordinal is 0xffffffff, then a list of all of the supported ordinals SHOULD be returned for the specified vendorId.

Output Parameters	Description
capabilityList	An array of PTS_Capability items.. Each Capability indicates the vendorId, the command ordinal and the implementation status for the requested commands.

6.7.12 PTS_ListSupportedAlgorithms

```
- generic request -

typedef struct {
      PTS_UInt32  cAlgs;  // count of supportedAlg returned
      PTS_AlgorithmId   algorithms[]; // array of PTS_AlgorithmId
} PTS_ListSupportedAlgorithms_Response
```

Description

This command is implemented by the PTS and is called by clients seeking to determine the algorithms supported by a PTS implementation.

The response lists the algorithms supported and currently usable by PTS.

Output Parameters	Description
cAlgs	Number of algorithms supported by the PTS implementation
algorithms[]	An array of `PTS_AlgorithmId`.

Error Codes	Condition

6.7.13 PTS_RegisterVerifyKey

```
typedef struct {
      PTS_Key     verifyKey;        // verification key
      PTS_UInt32  authSizeVk;
      PTS_Byte    authDataVk;
      PTS_Key     storeKey;         // storage key
      PTS_UInt32  authSizeSk;
      PTS_Byte    authDataSk;       // storage auth value
} PTS_RegisterVerifyKey_Request;

-Generic Response-
```

Description

This command is implemented by the PTS and is called by a management process.

The caller provisions PTS with an ID of a key that will be used by the PTS to verify signed objects.

authDataSk is used to access a protected storage system or to access a storage key identified by storeKey which can be used for integrity protecting the verifyKey. PTS should use the TPM key storage or TPM Non-Volatile storage capabilities if a TPM is available on the platform.

Verification keys are used by PTS to verify signatures on RIMM structures and policies.

Input Parameters	Description
`verifyKey.keyLength`	Length (in bytes) of the key
`verifyKey.keyId`	Identifier of the verify key. Contents of which is implementation specific. Normally this is a UUID.
`authSizeVk`	Length (in bytes) of authorization data for verifyKey
`authDataVk`	Authorization data for verifyKey
`storeKey.keyLength`	Length (in bytes) of the key
`storeKey.keyId`	Identifier of the storage key. Contents of which is implementation specific. Normally this is a UUID.
`authSizeSk`	Length (in bytes) of authorization data
`authDataSk`	Authorization data for accessing storage system

Error Codes	Condition
PTS_KEY_NOT_FOUND	Unable to find verify key
PTS_KEY_LENGTH_UNSUPPORTED	Singing key length not supported
PTS_CORRUPT_KEY	Key blob is corrupt
PTS_INVALID_PASS_PHRASE	Invalid pass phrase
PTS_AUTHENTICATION_FAILURE	Unable to authenticate to key storage device
PTS_INSUFFICIENT_RESOURCES	Not enough memory or persistent storage; or data structure limitation reached.

6.7.14 PTS_UnregisterVerifyKey

```
typedef struct {
      PTS_Key     quoteKey;
      PTS_Key     storeKey;         // storage key
      PTS_UInt32  authSizeSk;
      PTS_Byte    authDataSk;       // storage auth value

} PTS_UnregisterVerifyKey_Request;

-Generic Response-
```

Description

This command is implemented by the PTS and is called by a management process.

The caller specifies a key to be removed from the PTS list of registered verification keys.

Input Parameters	Description
verifyKey.keyLength	Length (in bytes) of the key
verifyKey.keyId	UUID of the verify key
storeKey.keyLength	Length (in bytes) of the storage key
storeKey.keyId	Identifier of the storage key. Contents of which is implementation specific. Normally this is a UUID.
authSizeSk	Length (in bytes) of authorization data
authDataSk	Authorization data for accessing storage system

Error Codes	Condition
PTS_KEY_NOT_FOUND	Unable to find key
PTS_KEY_LENGTH_UNSUPPORTED	Key length not supported
PTS_CORRUPT_KEY	Key object is corrupted
PTS_AUTHENTICATION_FAILURE	Unable to authenticate to key storage device

6.7.15 PTS_ListVerifyKeys

```
- generic request -
```

```
typedef struct {
      PTS_UInt32        numKeys;
      PTS_Key           verifyKeys; // list of registerd keys
} PTS_ListVerifyKeys_Response;
```

Description

This command is implemented by the PTS and is called by a client.

The set of currently registered verification keys is returned.

Output Parameters	Description
numKeys	The number of verifyKeys
verifyKeys	The currently registered list of verification keys

Error Codes	Condition
PTS_CORRUPT_KEY	A key object is corrupted

6.7.16 PTS_GetCookie

```
- generic request -

typedef struct {
      PTS_Cookie  cookie;  // PTS generated cookie value
} PTS_GetCookie_Response
```

Description

This command is implemented by the PTS and is called by clients to obtain a PTS_Cookie value that is unlikely to be in use by another session or thread. It is used to coordinate asynchronous responses to command invocations that support unsolicited command responses.

The client may call this command to be assured that a duplicate cookie value is not already in-use.

Identical cookie values MAY NOT be reissued within the same session.

Output Parameters	Description
cookie	A value that is highly unlikely to be in use by another command (thread)

Error Codes	Condition
PTS_DUPLICATE_COOKIE	No more cookie values can be supplied for the current session

7 Platform Bindings

As noted above, IF-PTS is a platform-independent interface. It is designed to support almost any platform. In order to ensure compatibility within a single platform, this section defines how IF-PTS is implemented on specific platforms.

7.1 Minimum Platform

This specification and these bindings do not support 16-bit platforms. Only 32-bit and higher platforms are supported.

7.2 32-Bit Platforms

PTS_UINT MUST be an unsigned integer of 32 bits length; PTS_UINT is assigned to be PTS_UInt32.

7.3 64-Bit Platforms

PTS_UINT MUST be an unsigned integer of 64 bits length; PTS_UINT is assigned to be PTS_UInt64.

7.4 Endian-ness

All data structures defined for use with PTS utilize big endian format. Big endian bit ordering follows the Internet standard and requires that the low-order bit appear to the far right of a word, buffer, wire format, or other area and the high-order bit appear to the far left.

PTS vendors have 3 implementation options with respect to endian-ness:

1) PTS utilizes big endian format only. This option is the minimum to implement. This option is the most simple, but has potential performance considerations on little endian platforms due to endian conversion.
2) PTS endian-ness is chosen at install time.
3) PTS vendor provides an extension to negotiate endian-ness.

7.5 Named Pipes

7.5.1 Windows Platform Configuration Details - Registry Key

A well-known registry key is used by the PTS to load configuration details. For Windows platforms, this key is defined within the HKEY_LOCAL_MACHINE hive as follows.

- HKEY_LOCAL_MACHINE
  - → Software
  - → Trusted Computing Group
  - → PTS
  - → COMMPIPE
  - → P, 0..n

[HKEY_LOCAL_MACHINE\SOFTWARE\Trusted Computing Group\PTS]

[HKEY_LOCAL_MACHINE\SOFTWARE\Trusted Computing Group\PTS\COMMPIPE]

"P0"="PTS-pipe-P0"

[HKEY_LOCAL_MACHINE\SOFTWARE\Trusted Computing Group\PTS\COMMPIPE\P0]

"pipename"="PTS-pipe-P0"

Each configuration key in PTS\COMMPIPE identifies an instance of PTS. I.e. P0 is the first PTS, P1 the second, etc.

Then, the actual pipe name (minus \\pipe\) is an attribute in each Pn. Additional attributes could then be added for each Pn.

Each configuration key contains an (unordered) set of values, as follows:

- the value *"Path"* is a REG_SZ String which contains the pathname of the component
- the optional value *"Description"* is a REG_SZ String which contains a vendor-specific human-readable description of the IMC DLL

The name and description are for ease of administration and may be ignored by PTS, except for human interface purposes; only the Path data matters. Additional values or keys may be present within the keys listed above. PTS MUST ignore unrecognized values and keys.

An extension mechanism has been defined so that vendors can place vendor-specific keys or values in the PTS key or any subkey without risking name collisions. The name of such a vendor-specific key or value must begin with the vendor ID of the vendor who defined this extension. The vendor ID must be immediately followed in the name by an underscore which may be followed by any string.

7.5.2 UNIX/Linux Platform Configuration Details

Implementations of PTS on UNIX and Linux operating systems will need access to configuration details specifying the location of some data (such as named pipes for communication) and other details that are not easily specified in a document such as this in a way that will be consistent with the platform's own rules or administrator's preferences. Any configurable information that the PTS needs to find should be put into /etc/pts_config.

7.5.2.1 Format of /etc/pts_config

The /etc/pts_config file specifies configurable information that a PTS may need to access. This file is only required if the PTS needs to access configuration details that are not already known by other means (hard-coded, command-line arguments, etc).

The /etc/pts_config file is a UTF-8 file. If a PTS encounters a character that is not US-ASCII and the PTS can not process UTF-8 properly, the PTS SHOULD indicate an error and not load the file at all. In fact, the PTS SHOULD respond to any problem with the file by indicating an error and not loading the file at all. All characters specified here are specified in standard Unicode notation (U+nnnn where nnnn are hexadecimal characters indicating the code points).

The /etc/pts_config file is composed of one or more lines. Each line ends in U+000A. No other control characters (characters with the Unicode category Cc) are permitted in the file. A line that begins with U+0023 is a comment. All other characters on the line should be ignored. A line that does not contain any characters should also be ignored.

The /etc/pts_config file MUST not contain more than one attribute with the same human-readable name. A PTS that encounters such a file SHOULD indicate the error and MAY not load the file at all.

Here is a specification of the file format using ABNF as defined in RFC 2234 [1]:

pts_config = *line
line = *(comment / empty / pipedef / avpair) CRLF comment = %x23 *(VCHAR / WSP) empty = ""
pipedef = commpipe 1*WSP *WSP value
commpipe = %0x43.4F.4D.4D.50.49.50.45 ; COMMPIPE in caps avpair = name 1*WSP *WSP value name = *(DIGIT / ALPHA / “-”) value = *(%0x24-7E) ; '$' - '~'

Here is a sample file specifying the “COMMPIPE” value which is defined as /var/tmp/PTS-pipes/P0:

Simple PTS config file
COMMPIPE /var/tmp/PTS-pipes/P0

7.5.2.2 Required Entries

A PTS configuration file MUST include the “COMMPIPE” attribute and an associated value appropriate for the platform. The value associated with this attribute may vary depending on the platform or administrator preferences for that platform.

7.5.2.3 Other Entries

PST implementers MAY choose to support additional attribute-value pairs in the PTS configuration file. These additional parameters MUST follow the format rules defined above

8 Security and Privacy Considerations

8.1 Security Considerations

IF-PTS defines the interface to a security service, PTS. PTS is used to provide a trust service for other components. Architectural assumptions for PTS are defined in §6.3 and §6.4 of the TNC Architecture, ref [1]. The trust service provided by PTS is dependant on a Root-of-Trust mechanism and a transitive trust mechanism for the platform.

PTS services and functionality is exposed to host processes through the IF-PTS interface. PTS requirements are documented in §2.3.2 of this document and PTS assumptions are detailed in §2.4 of this document.

PTS is dependant on the underlying platform-specific IPC mechanism to secure the interface.

If clientId has already been used to initialize a concurrent session, PTS_Initialize will fail and the caller should select a different value for clientId. It may be possible for a rogue to guess clientId and sessionName by testing for namespace collisions.

8.2 Privacy Considerations

The TPM supports privacy mechanisms to protect EK (Endorsement) Public keys from disclosure.

The PTS relies on the IMC collecting its information to vet the data collected to determine which data are disclosed.

In addition, the TPM automatically takes some measurements itself upon initialization without requests via IF-PTS. These measurements are either built into the PTS implementation and / or configured by the PTS administrator. The PTS vendor SHOULD state what is measured and reported automatically by PTS without configuration.

9 Sequence Diagrams

The sequence diagrams in this section describe typical usage of PTS interfaces. Sequence messages are exchanged between the following Objects:

- PTS – Platform Trust Service – a system service that implements PTS functionality.
- PTS-IMC – a process interacts with the PTS service specifically to report measurements taken directly by PTS.
- IMC – a dynamic library that plugs into the TNC-Client and may interact with the PTS service.
- Component – a generic process on which PTS computes integrity measurements, but does not open a session with PTS.
- TPM – Trusted Platform Module – a trusted device that maintains integrity state in a Platform Configuration Register (PCR) and protects integrity measurements for remote consumption using digital signatures.
- TNCC -TNC Client – a process that exchanges integrity measurements with a TNC Server.
- System – the operating system or operating environment.
- PTS Log – a.k.a. Integrity Measurement Log which is the logging mechanism used by the PTS.
- NAR - Network Access Requestor – a service or driver that maintains a connection to network access equipment.

9.1 Component Scan

The following sequence diagram assumes a RIMM structure for the component to be measured has previously been registered with PTS.

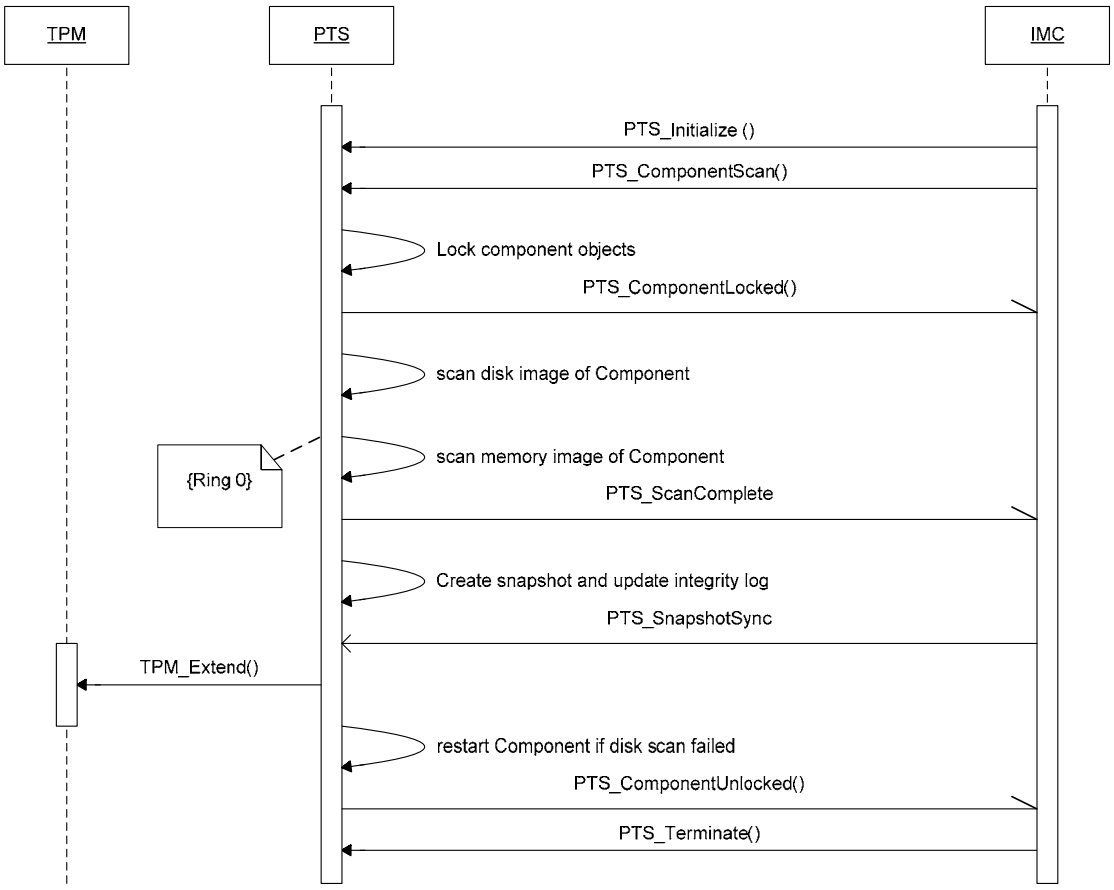


Figure 4 - Component Scan Sequence

The component scan sequence shows two types of integrity anomaly detection techniques; a scan of a binary image on disk and a scan of a binary image in memory. Control of the scan behavior is directed by an IMC process where the target of the scan is a passive participant.

Steps:

a. The IMC opens a session with the PTS.

b. The IMC instructs PTS to perform a scan of a Component in the system.

c. PTS locks the Component binary files on disk to prevent inadvertent update while the files are scanned. Note: if only a memory scan is performed it is not necessary.

d. PTS notifies the IMC when files are locked to synchronize IMC actions on the Component.

e. PTS performs either a disk or memory scan or both as determined by the IMC. Note: If a scan of the on-disk image fails unexpectedly, it is recommended that remediation action be taken regardless of a successful memory scan.

f. PTS notifies the IMC when the scan has completed. Note: It is not necessary to return the PTS_ComponentLocked or PTS_ComponentUnlocked asynchronous messages if only memory scan is employed.

g. PTS creates a snapshot and updates internal state and log files as needed.

h. IMC instructs PTS to protect the integrity of the scan result by extending the hash of the scan result into a TPM PCR.

i. PTS extends the hash of the scan result into a TPM PCR.

j. PTS removes file locks and notifies the IMC that the component is available for loading or possible remediation.

k. IMC may issue additional PTS commands or terminate the session.

When doing component scans the PTS must compute measurements that match reference measurements. PTS may use RIMM structures as a guide to identifying relevant elements and to identify components using ComponentID. PTS must be able to locate component element installation locations. The RIMM may provide relative pathnames, but fully qualified pathnames requires consultation with the system configuration database.

In particular, PTS may be configured to automatically check TNC components upon system startup. This can be achieved using a variety of vendor specific techniques ranging from vendor extensions for registering the check or through PTS configuration settings.

9.2 Snapshot Creation

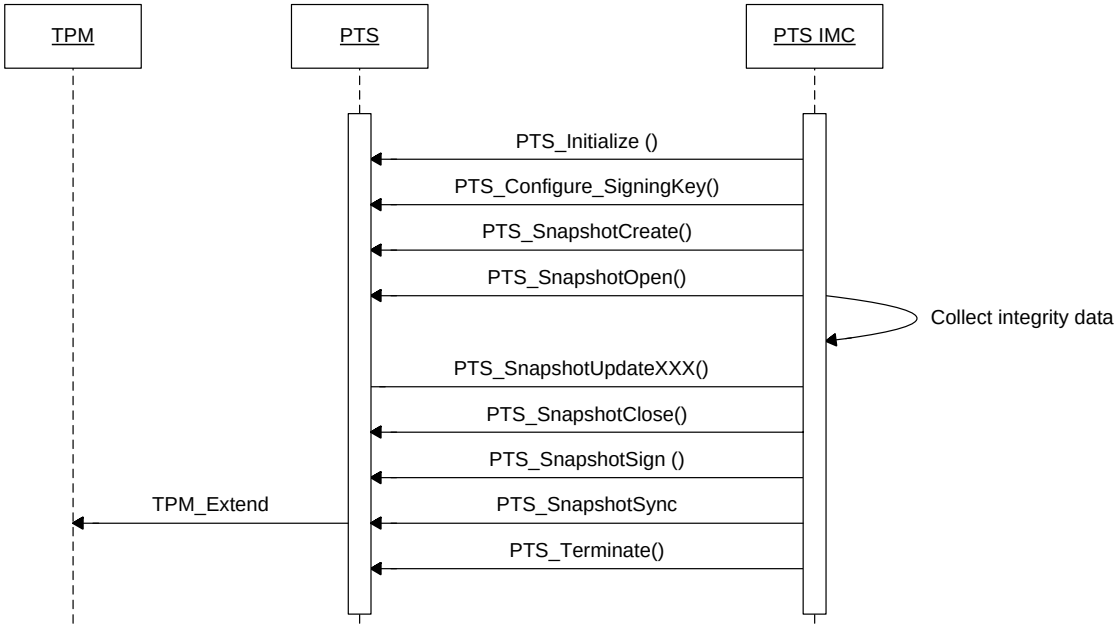


Figure 5 - Snapshot Creation Sequence

Snapshot structures are the building block elements of an Integrity Report. Each component of a subsystem is described by a snapshot having a component ID corresponding to the component for which an IMC has collected integrity data. The PTS Client may use PTS to format integrity data for use by remote entities such as an IMV. PTS may also be used to affix a digital signature to a snapshot, make a snapshot persistent and to synchronize the snapshot integrity state with a TPM protected integrity log.

A typical sequence of operations is described in the following steps.

Steps:

1) The IMC opens a session with the PTS.
2) At some time prior to the use of PTS_SnapshotSign, a signing key should be configured into PTS using PTS_Configure_SigningKey.
3) A blank (unpopulated) snapshot structure is created using PTS_SnapshotCreate. The snapshot is given a unique identifier that disambiguates snapshot instances.
4) The snapshot can be opened for update using PTS_SnapshotOpen. A descriptor to the snapshot context is returned.
5) An open snapshot descriptor is used to update various elements of a snapshot using the PTS_SnapshotUpdateXXX interfaces. Several interfaces are defined for each section of the snapshot.
6) To close the snapshot to updates use PTS_SnapshotClose.
7) A snapshot can be digitally signed using PTS_SnapshotSign. A key supplied by PTS_Configure_SigningKey is used to add a signature.
8) An existing (closed) snapshot can be synchronized to protect the integrity of the snapshot by extending the snapshot composite hash into a TPM PCR using TPM_Extend.
9) PTS Client may issue additional PTS commands or terminate the session.

9.3 Report Specification and Generation

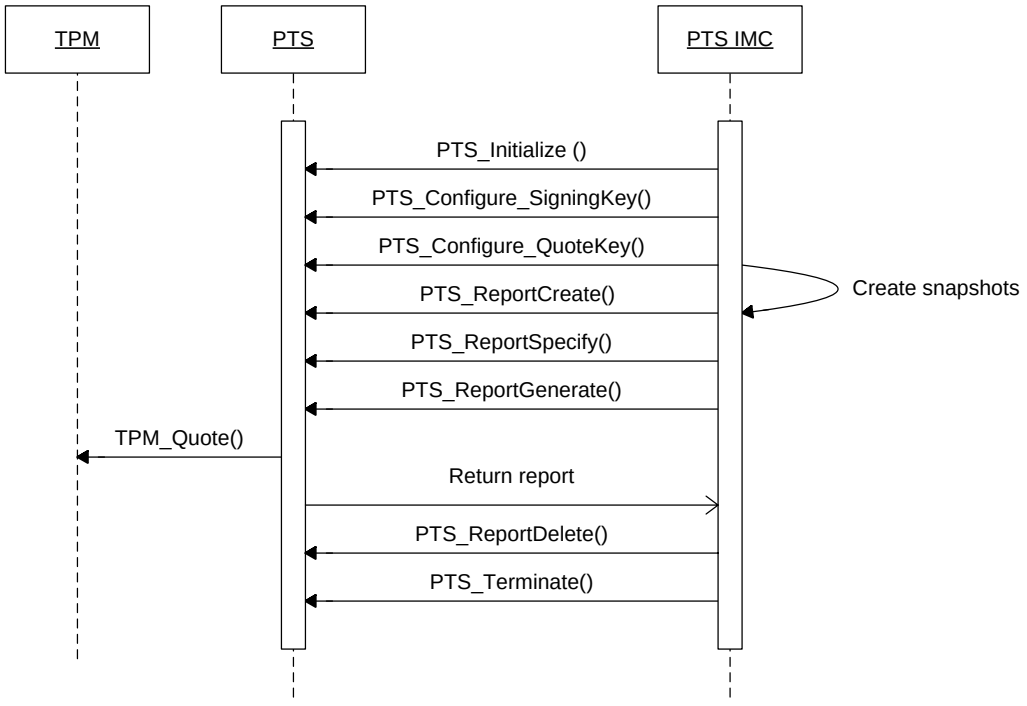


Figure 6 - Report Generation Sequence

An integrity report can be generated by PTS containing multiple snapshot structures. TPM integrity protections may be included also. An integrity report is useful for capturing the integrity state of more complex structures such as the platform's transitive trust path, a collection of related components or a series of measurements taken over a period of time.

A typical sequence of commands follows.

Steps:

1) The PTS Client opens a session with the PTS.
2) PTS_ConfugureSigningKey and PTS_ConfigureQuoteKey are called anytime prior to PTS_ReportGenerate to configure keys for signing and for TPM quote commands.
3) Integrity reports primarily consist of snapshots. Therefore, the snapshots intended for the report must be generated or imported into PTS.
4) PTS_ReportCreate generates an empty report context. A report identifier is used to refer to the report.
5) The contents of the integrity report are specified using PTS_ReportSpecify. Many report dimensions are possible.
6) Report contents although marked for inclusion, are not committed to the report until PTS_ReportGenerate is called.
7) If a TPM is used, the report may include TPM PCR values and TPM applied digital signature using TPM_Quote command.
8) The signature is applied prior to completing PTS_ReportGenerate computation. The report is returned to the PTS Client.
9) PTS Client may issue additional PTS commands or terminate the session.

9.4 Rule Evaluation

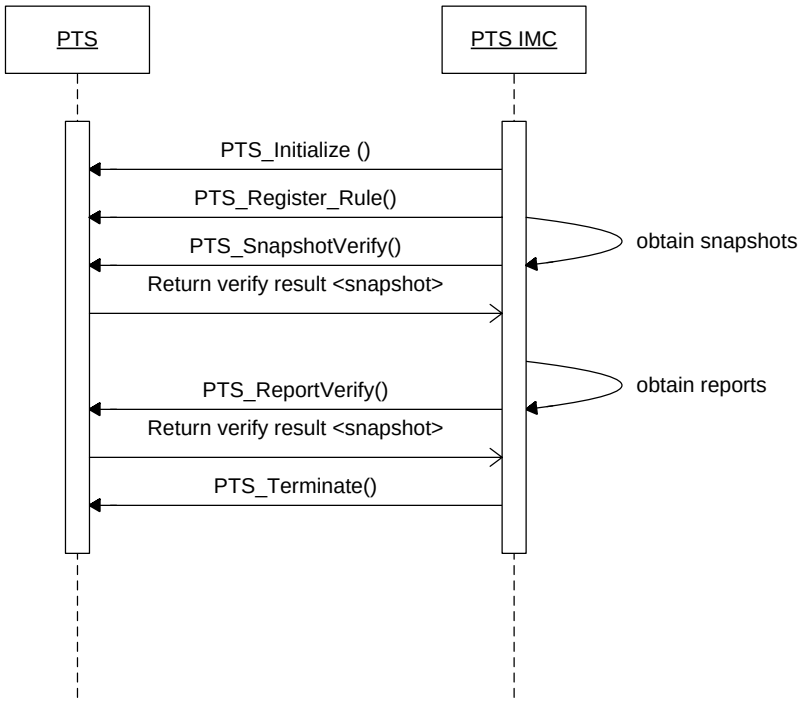


Figure 7 - Rule Evaluation Sequence

The PTS process can aid in verification of collected measurements if formatted as a snapshot or integrity report. PTS verification applies a verification policy as specified by rule structures that are configured into PTS prior to invocation of PTS_ReportVerify or PTS_SnapshotVerify. The rule provides reference integrity values and other logic necessary to compare a snapshot or integrity report to expected values.

Verification will generate a new snapshot containing the results of an applied rule. The new snapshot could be reported in place of an otherwise unevaluated snapshot or report. Verification by PTS allows larger data sets to be evaluated locally and helps reduce bandwidth requirements for remote verification.

Steps:

1) The PTS Client opens a session with the PTS.
2) A repository of rules that the PTS may use is configured sometime prior to PTS_SnapshotVerify or PTS_ReportVerify using PTS_Register_Rule. PTS will make verification rules persistent for later use. Rules should be cryptographically bound to the PTS service to prevent rogue insertion of unauthorized rules.
3) Prior to verification, the snapshots to be verified must be available to PTS.
4) Verification operations are applied when PTS_SnapshotVerify is called.
5) Results of the verification are placed into a snapshot that is generated by PTS and reported in the form of a snapshot.
6) Verification policy is applied when PTS_ReportVerify is called.

If the rule is bound or sealed to the PTS, TPM_UnBind or TPM_Unseal is used to obtain a decryption

7) PTS Client may issue additional PTS commands or terminate the session.

9.5 Snapshot Synchronization

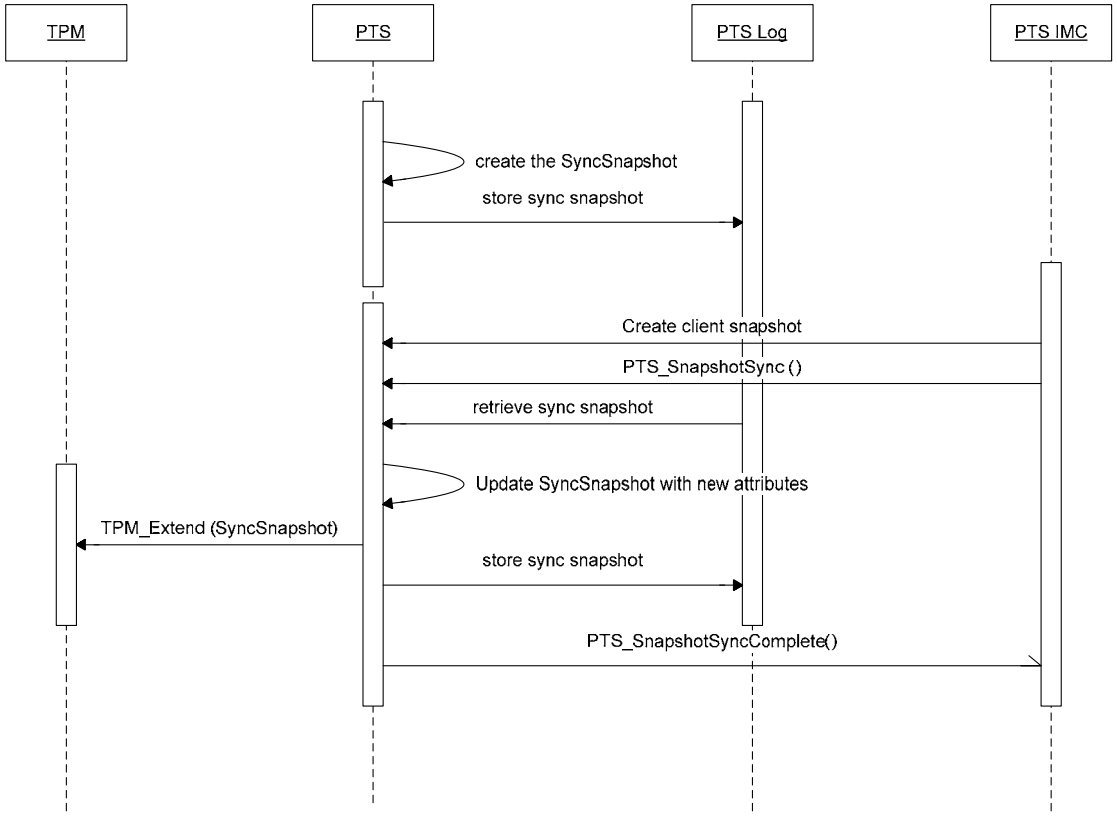


Figure 8 - Snapshot Synchronization Sequence

The PTS service can protect collected measurements using a TPM by extending the composite hash of a snapshot into a TPM PCR. PTS can coordinate synchronization of multiple snapshots through a special snapshot known as the *sync* snapshot. The SyncSnapshot captures a composite hash of all snapshots that are extended into a TPM PCR.

PTS is responsible for maintaining a persistent log of collected integrity values. The sync-snapshot can be used to maintain history of TPM PCRs that have been reset.

One possible approach for synchronization of snapshots with a TPM is described below.

Steps:

1) The PTS Client opens a session with the PTS.
2) As a prerequisite, snapshots must be made available to PTS through PTS_SnapshotCreate or PTS_SnapshotImport.
3) PTS may make snapshots persistent according to internal policy.
4) The PTS Client schedules a snapshot for synchronization through PTS using the PTS_SnapshotSync command.

5) PTS computes a composite hash of a sync-snapshot and includes the snapshot composite hash supplied by PTS Client.
6) The supplied snapshot composite hash is extended into a TPM PCR such that the resulting value exactly matches the resultant composite hash of the sync-snapshot.
7) The sync-snapshot is stored for later use.
8) The PTS Client is notified of the status of synchronization request asynchronously using PTS_SnapshotSyncComplete.
9) PTS Client may issue additional PTS commands or terminate the session.

9.6 Example Usage Scenarios

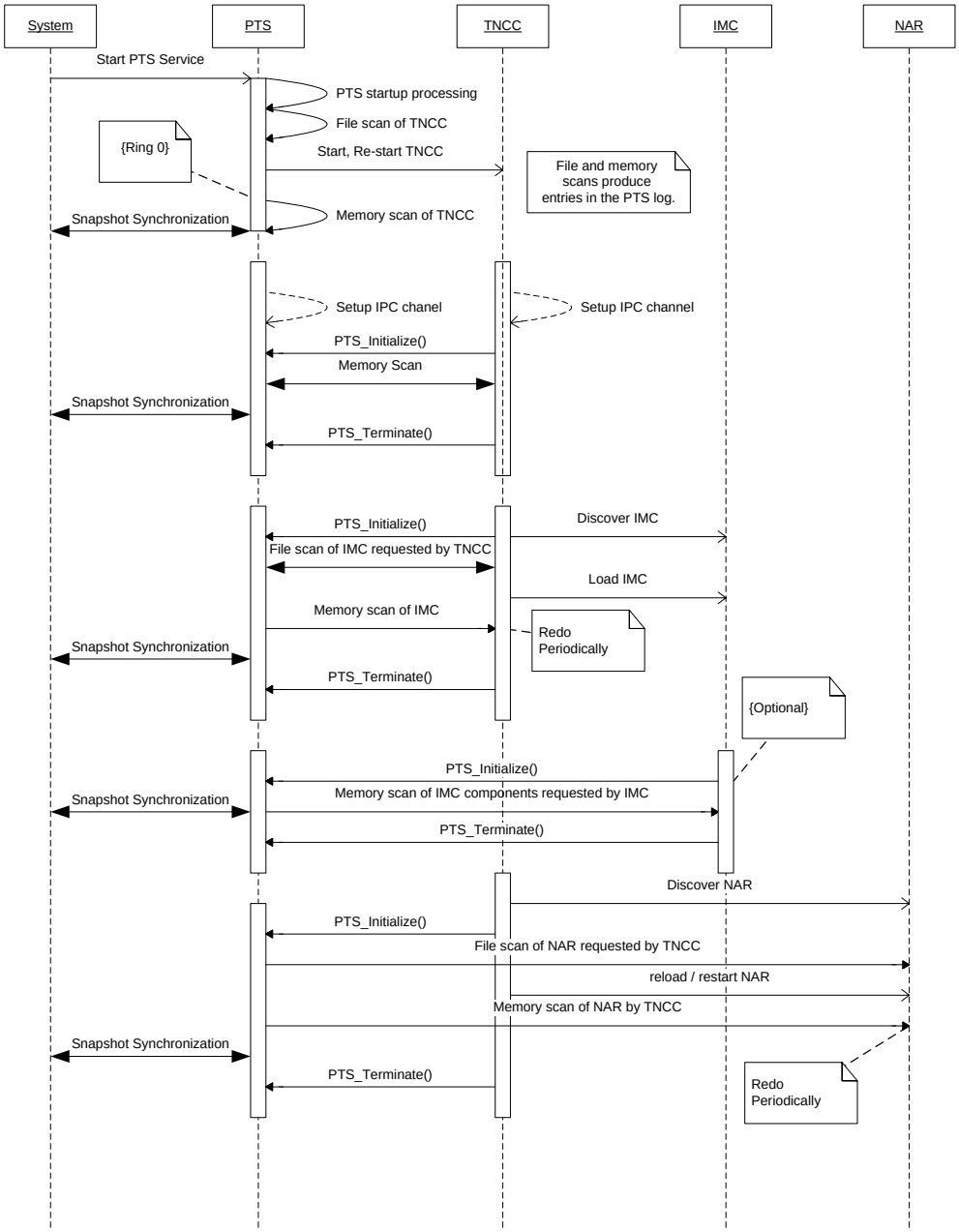


Figure 9 - Example Scenarios

The sequence diagram in the above diagram shows several possible scenarios involving PTS. In this diagram, the “System” entity combines both the TPM and the operating system of the platform. These entities were combined purely due to lack of space in the diagram and does not imply any logical or physical combination of the two. The operating system starts the PTS service. Snapshot synchronization occurs via a call to TPM Extend. These examples are provided as a reference for typical deployment and usage.

10 Usage Scenarios

10.1 Establishing TNC Subsystem Integrity

In this scenario the integrity of the TNC subsystem (IMC, IMV, TNC-Client, TNC-Server and the PTS itself) are integrity scanned. A subsystem integrity report is prepared by a PTS-specific IMC hereafter referred to as the *PTS-IMC*. The integrity report is communicated to a verifier, hereafter referred to as the *PTS-IMV* to be evaluated and factored into the network access control decision.

The primary motivation of this use case is to improve the degree of assurance that the TNC mechanism itself has not been compromised. Compromise of collectors, TNC-Client, network access components or the PTS itself could cause a network access control decision to be called into question.

The methodology for determining integrity calls for an initial calculation of a baseline measurement (cryptographic hash) of TNC subsystem components. The initial calculation is called the *baseline measurement* and should be computed by a trusted process (such as a manufacturers build process). The baseline measurement is stored in a secure location and is consulted by a verifier such as the PTS-IMV.

One or more integrity measurements of TNC subsystem components are calculated at critical points during the operation of the subsystem. This scenario identifies three critical points of operation when an integrity measurement should be calculated. Other points of operation may be reasonably considered, this usage considers the following:

- Program Load – If a program is modified on disk the change may be considered benign up to the point that the program code is loaded into memory. Computing a measurement at load time establishes the integrity of the on-disk image before being subject to threats associated with the runtime environment.
- Service Invocation – Services typically startup automatically and stay running as long as the system is up. Clients of the service may want to have the integrity state of the service calculated prior to using the service. It makes most sense to perform the calculation just prior to client use to minimize the window between possible compromise and client use.
- IMC Reporting – The point when an IMC is ready to report is the point in which the most current state may be reported. Collecting subsystem integrity measurements of reporting IMCs allow a verifier to assess the condition of the reporting infrastructure upon which the IMC report is prefaced.

Calculation of integrity measurements at the intended point involves some logistics. The component being scanned must be identified and accessed. The scan result must be stored and/or prepared for transfer. The *platform trust service* (PTS) helps with those logistics. The PTS performs the integrity measurement calculations for executable files on a storage device. A sequential byte-by-byte digest computation may be sufficient. The PTS may calculate a digest of program code resident in memory or in paged memory. A naïve hash calculation of a single monolithic code segment may not be possible in most environments. Typically code relocations result in many discrete code segments being created out of a single binary image on disk. This use case presumes memory resident code can be measured, but does not attempt to describe what might otherwise be considered state-of-the-art consistency checking techniques for executing code.

The PTS interacts with TNC components as a system service. TNC components can control when integrity checks are performed on it or other components, but they do give permission to PTS to perform scanning operations.

Trust of the PTS depends on other integrity protection mechanisms built into the platform. For the purposes of this use case the PTS anticipates system firmware and or kernel extension exists that will scan the PTS code. Therefore a “transitive trust” path can be derived from the TPM root-of-trust-for-measurement and can be included in the PTS-IMC integrity report.

The PTS-IMC performs several duties. It retrieves integrity values collected by the PTS. It assembles evidence of a transitive-trust chain and ensures the evidence is expressed in an interoperable format. PTS-IMC anticipates the verification steps of a PTS-IMV.

10.1.1 Collection

An objective of collection is to construct a model of the environment in which TNC components operate. The completeness of the model and the veracity of evidence establish the level of assurance that may be associated with reports of other IMCs and the level of trust in the channel.

10.1.1.1 Pre-OS Boot

Antecedents of platform integrity may begin before an operating system is loaded and even before firmware executes. Measurement code that is interwoven into the pre-OS environment can describe the boot state and store it in registers that are made available after OS initialization. If a history of pre-OS states cannot be recorded in a log, the measurement values themselves will become well-known values that are integrated into verification policies.

One important measurement performed during the pre-OS operation is the measurement of the initial program loader or boot image. This measurement allows a transitive-trust link to be extended across the pre-OS environment into the OS environment.

10.1.1.1.1 Example of Pre-OS Measurements

Consider an EFI-based personal computer containing a TPM (see Figure 10). There is code in the BIOS boot block called the Root-of-Trust for Measurement (RTM) that runs before other code. The RTM uses the TPM to extend a hash of the POST code into a TPM Platform Configuration Register (PCR). The RTM is trusted code because no other code measures it and execution begins with the RTM.

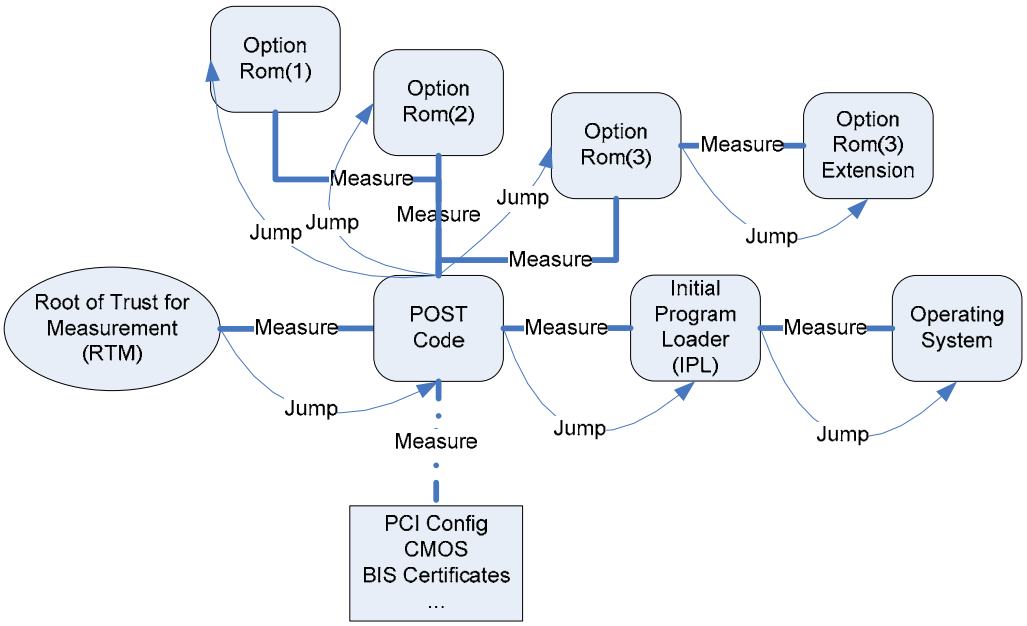


Figure 10 - EFI BIOS Measurements and Tansitive-Trust

The RTM computes the hash of the next code to run, POST code and extends a TPM PCR. The execution thread jumps to the POST code. POST code measures optional ROMs and executes them. In this example option ROM(3) can be decompressed and measured by another segment of the option ROM(3). Theoretically, any number of option ROM segments may be decompressed, measured and executed. Eventually program flow returns to the POST code and other data such as PCI configuration, CMOS, certificates etc… can be measured. Prior to POST code completing it measures the Initial Program Loader (IPL), extends a PCR and transfers

execution to the IPL. The IPL determines an appropriate OS to load, measures it, loads it and then allows the OS code to take control.

A record of the pre-OS boot sequence may be stored in an ACPI table for later review. The PCRs can be used to validate that a measurement log (in the ACPI table) is consistent. The execution path (sequence of jumps) is called the *transitive-trust chain*.

10.1.1.2 Pre-PTS Startup

Prior to the PTS service starting the operating system will have control of system resources including disk and memory. The PTS service is measured by a trustworthy measurement thread active in the post-OS environment. This may be a protected system management thread, virtual machine monitor or other form of isolated execution.

The PTS (drivers and program code) are digested. The digest result along with other identifying information such as its manufacturer, version and patch level may be extended into a TPM PCR and accompanying log may be used for verification at a later time. Tampering of the log is detected by comparing the PCR value to a similar value computed using the log entries. Other protection mechanisms may be applied but are out of scope for this use case.

10.1.1.3 PTS Operation

The PTS computes measurements for TNC components and records a history of measurements for later review. The PTS may apply multiple strategies to detect and prevent tampering. Two techniques for tamper detection are computing a message digest of the program image while on disk and computing a message digest of the program image after it is loaded into system memory.

The PTS is responsible for collecting and maintaining measurements for all TNC components. These include IMCs, TNC-Clients and other components that may play a role in network access control collection and reporting infrastructure.

The PTS can render measurements into an interoperable format suitable for any IMC / IMV to consume.

10.1.1.4 PTS-IMC Collection

The PTS-IMC constructs the transitive-trust chain of evidence and reports it to a corresponding PTS-IMV. The PTS is its primary resource for obtaining the chain of evidence. It may implement protocols for reporting and synchronizing with an IMV, the collected transitive-trust and TNC subsystem state information.

10.1.1.5 Collection at the Time-of-Manufacture

Some attributes of the platform cannot be collected directly from the platform. For example, knowledge that a motherboard may support memory, disk or CPU virtualization may not be assessed directly from a platform. Furthermore, quality measures intrinsic to a manufacturing process are not directly observable. For this class of information an out-of-band collection facility is employed. Out-of-band implies an alternate infrastructure (other than that provided by TNC defined components) is needed. These measurements are made available to the PTS-IMV through a PTS-IMV backend service.

10.1.2 Reporting

The PTS-IMC and PTS-IMV cooperate in the execution of protocols for requesting and reporting measurements. Protocols may range in complexity and may be vendor specific or standard. In this usage a protocol that reports the transitive trust chain is assumed. The PTS-IMV may request the PTS-IMC to send the transitive-trust chain of elements in a single message or may incrementally request the elements. The PTS-IMC may supply the transitive-trust chain without solicitation from the PTS-IMV.

10.1.2.1 Followup Reports

The PTS-IMV and PTS-IMC may further define a protocol that authenticates PTS-IMC messages. This protocol is targeted for use as a "watch-dog" process that alerts the PTS-IMV of configuration change without reporting significant details about the transitive-trust chain.

10.1.3 Evaluation

The PTS-IMV evaluates messages sent by PTS-IMC. PTS-IMV also receives measurements from one or more services in the network or Internet regarding assertions of integrity or quality that are not observed by the PTS or some other agent on the client platform.

The PTS-IMV also receives policy statements from the network administrator. The policy describes acceptable and unacceptable transitive-trust configurations. The policies may be expressed in proprietary formats or in industry accepted formats such as XACML[2], SAML[3] or XRML[4].

If a watch-dog protocol is used, the PTS-IMV evaluates the result by verifying watch-dog packets in accordance with the decryption keys and expected values.

A determination is made recommending a course of action. Possible actions may be deny access, allow access or allow access with restrictions. In the case of denied access or in the case of allowed access with restrictions, it may be appropriate to recommend a remediation action. In the case of any result to allow, a weighted value may be included that indicates non-binary confidence of operational consistency of the client platform.

The evaluation results are communicated to the TNC-Server or alternate for further evaluation.

10.1.4 Decision Making

Decision making follows the model established by IF-IMV and IF-TNCCS. The PTS-IMV provides its evaluation result (along with all other IMVs) to the TNC-Server (or appropriate other decision making service) to be factored together to arrive at a singular decision. The decision could be to allow access, deny access or allow access with restrictions.

10.1.5 Remediation

In cases where the PTS-IMV recommends a remediation action, the PTS-IMC responds by starting a remediation protocol with an appropriate service. Examples of possible remediation actions for PTS-IMC include the following:

- Registration of a public key used to authenticate measurement reports
- Registration with local services, drivers or subsystems upon which the PTS depends for its position in the transitive-trust chain
- Update of PTS or PTS-IMC binaries and configuration files
- Update of any measurement agent in the transitive-trust chain
- Update of policies used in the formation of a watch-dog event
- Download of credentials or other out-of-band measurements for inclusion in PTS reports

The PTS-IMC may solicit the aid of the PTS or other platform services to perform remediation actions. If remediation support exists in the PTS-IMV and restrictions of the underlying network access control channel permit; the PTS-IMC may engage in a remediation protocols with PTS-IMV.

[2] http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf

[3] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security#samlv20

[4] http://www.xrml.org/

10.2 Anti-Virus Integrity Reporting

This scenario follows a collection process performed by an "AV-IMC" that collects details about an AV engine and its attributes. The AV-IMC interacts with the PTS service to render the AV integrity values in an interoperable format and to save the collected values in a log. The log is protected from tampering using the TPM. The "AV-IMC" interacts with an AV-IMV (through the TNCC/TNCS) who evaluates the collected values and makes an access control recommendation. It may be necessary for the AV-IMV to consult a server-side PTS service that parses formatted integrity values.

11 References

[1] Trusted Computing Group, TNC Architecture for Interoperability, Specification Version 1.1, Revision 1.0, April 2006.

[2] Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, Internet Engineering Task Force RFC 2119, March 1997.

[3] ISO, ISO/IEC 9899:1999, Programming Languages – C, 1999.

[4] XML Schema Specification, http://www.w3.org/XML/Schema.

[5] Crocker, D., P. Overell, “Augmented BNF for Syntax Specifications: ABNF”, Internet Engineering Task Force RFC 2234, November 1997.

[6] Trusted Computing Group, IWG Core Integrity Manifest Schema Specification v1.0, 2006.