TPM Main
Part 1 Design Principles

Specification Version 1.2
Level 2 Revision 103
9 July 2007
Published

Contact: tpmwg@trustedcomputinggroup.org

TCG Published

Copyright © TCG 2003 - 2007

TCG

Copyright © 2003-2007 Trusted Computing Group, Incorporated.

Disclaimer

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

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

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

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

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

Copyright © TCG

Acknowledgement

TCG wishes to thank all those who contributed to this specification. This version builds on the work published in version 1.1 and those who helped on that version have helped on this version.

A special thank you goes to the members of the TPM workgroup who had early access to this version and made invaluable contributions, corrections and support.

David Grawrock

TPM Workgroup chair

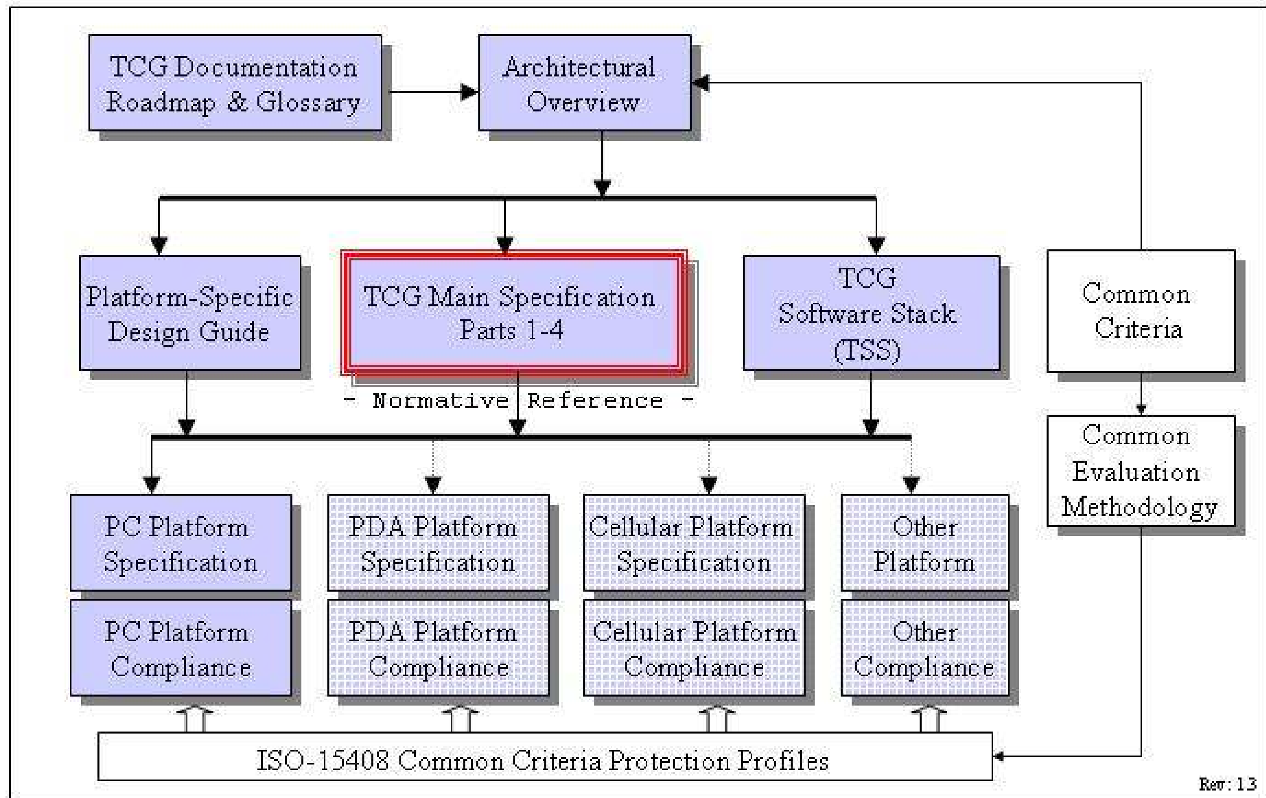
Change History

Version	Date	Description
Rev 50	Jun 2003	Started 30 Jun 2003 by David Grawrock First cut at the design principles
Rev 52	Jul 2003	Started 15 Jul 2003 by David Grawrock Moved
Rev 58	Aug 2003	Started 27 Aug 2003 by David Grawrock All emails through 28 August 2003 New delegation from Graeme merged
Rev 62	Oct 2003	Approved by WG, TC and Board as public release of 1.2
Rev 63	Oct 2003	Started 2 Oct 2003 by David Grawrock Kerry email 7 Oct “Various items in rev62” kerry email 10 Oct “Other issues in rev 62” Changes to audit generation
Rev 64	Oct 2003	Started 12 Oct 2003 by David Grawrock Removed PCRWRITE usage in the NV write commands Added locality to transport_out log Disable readpubek now set in takeownership. DisableReadpubek now deprecated, as the functionality is moot. Oshrats email regarding DSAP/OSAP sessions and the invalidation of them on delegation changes Changes for CMK commands. Oshrats email with minor 63 comments
Rev 65	Nov 2003	Action in NV_DefineSpace to ignore the Booleans in the input structure (Kerry email of 10/30 Transport changes from markus 11/6 email Set rules for encryption of parameters for OIAP,OSAP and DSAP Rewrote section on debug PCR to specify that the platform spec must indicate which register is the debug PCR Orlando FtF decisions CMK changes from Graeme
Rev 66	Nov 2003	Comment that OSAP tied to owner delegation needs to be treated internally in the TPM as a DSAP session Minor edits from Monty Added new GetCapability as requested by PC Specific WG Added new DP section that shows mandatory and optional Oshrat email of 11/27 Change PCR attributes to use locality selection instead of an array of BOOL’s Removed transport sessions as something to invalidate when a resource type is flushed. Oshrat email of 12/3 added checks for NV_Locked in the NV commands Additional emails from the WG for minor editing fixes
Rev 67	Dec 2003	Made locality_modifier always a 1 size Changed NV index values to add the reserved bit. Also noticed that the previous NV index values were 10 bytes not 8. Edited them to correct size. Audit changes to ensure audit listed as optional and the previous commands properly deleted Added new OSAP authorization encryption. Changes made with new entity types, new section in DP (bottom of doc) and all command rewritten to check for the new encryption
Rev 68	Jan 2004	Added new section to identify all changes made for FIPS. Made some FIPS changes on creating and loading of keys Added change that OSAP encryption IV creation always uses both odd and even nonces Added SEALX ordinal and changes to TPM_STORED_DATA12 and seal/unseal to support this
Rev 69	Feb 2004	Fixup on stored_data12.

 Copyright © TCG

		Removed magic4 from the GPIO Added in section 34 of DP further discussion of versioning and getcap DP todo section cleaned up Changed store_privkey in migrate_asymkey Moved text for getcapabilities – hopefully it is easier to read and follow through on now.
Rev 70	Mar 2004	Rewrite structure doc on PCR selection usage. New getcap to answer questions regarding TPM support for pcr selection size
Rev 71	Mar 2004	Change terms from authorization data to AuthData.
Rev 72	Mar 2004	Zimmermann's changes for DAA Added TPM_Quote2, this includes new structure and ordinal Updated key usage table to include the 1.2 commands Added security properties section that links the main spec to the conformance WG guidelines (in section 1)
Rev 73	Apr 2004	Changed CMK_MigrateKey to use TPM_KEY12 and removed two input parameters Allowed TPM_Getcapability and TPM_GetTestResult to execute prior to TPM_Startup when in failure mode
Rev 74	May 2004	Minor editing to reflect comments on web site. Locked spec and submitted for IP review
Rev 76	Aug 2004	All comments from the WG Included new SetValue command and all of the indexes to make that work
Rev 77	Aug 2004	All comments from the WG
Rev 78	Oct 2004	Comments from WG. Added new getcaps to report and query current TPM version
Rev 82	Jan 2005	All changes from emails and minutes (I think).
Rev 84	Feb 2005	Final changes for 1.2 level 2
Rev 88	Aug 2005	Eratta level 2 release candidate
Rev 91	Sept. 2005	Update to Figure 9 (b) in section 9.2 by Tasneem Brutch
Rev 100	May 2006	Clarified CTR mode
Rev 101	Aug 2006	Added deactivated rationale. Clarified number of sessions. Changed "set to NULL" to "set to zero". Added NV index D bit rationale. Added _INFO key rationale and clarified cases where _INFO keys act as _SHA1 keys.
Rev 102	Sept 2006	Minor typos only. No functional changes.
Rev 103	Oct 2006	Note that blobs encrypted in blocks must have integrity chaining. Merged two AIK sections. Self-test checks EK using encryption, not signing.

TCG Doc Roadmap – Main Spec



Copyright © TCG

TCG Main Spec Roadmap

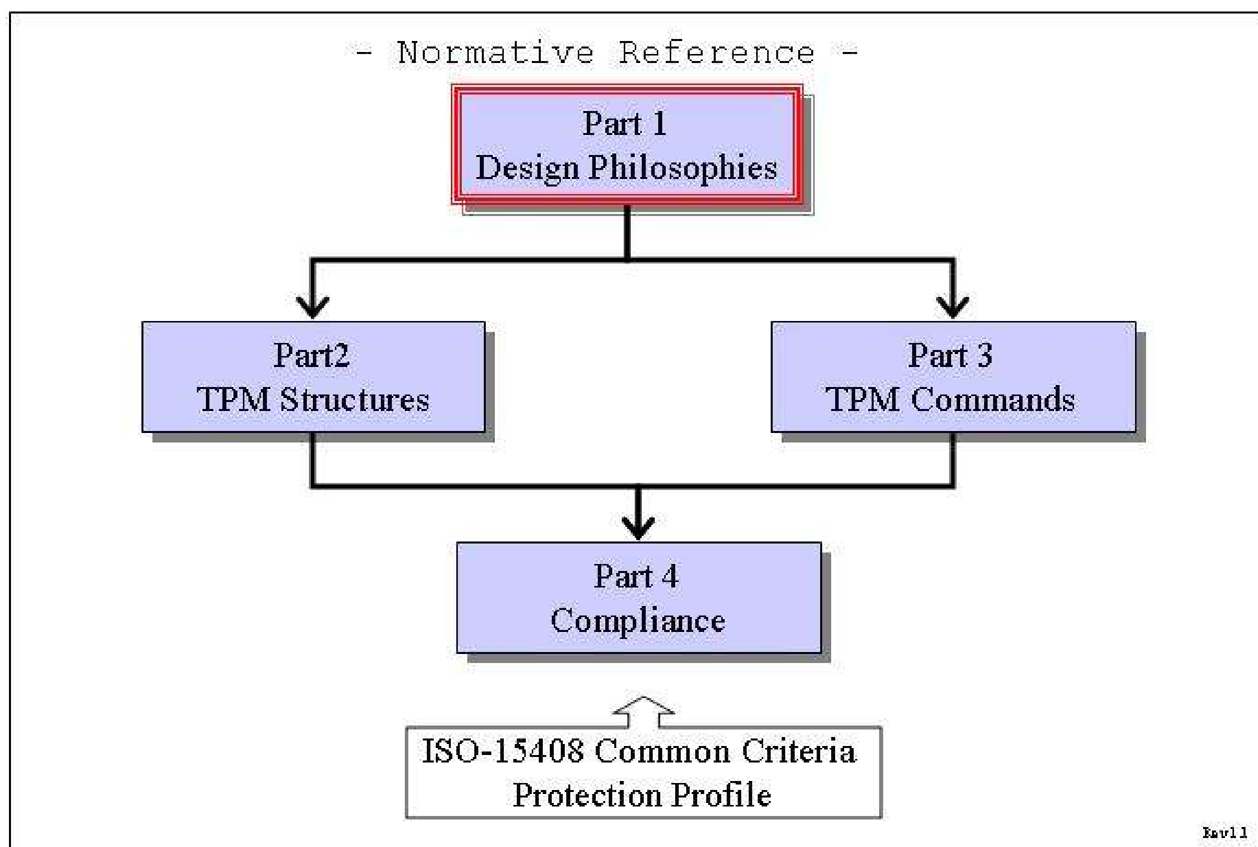


Table of Contents

1. Scope and Audience 1
1.1 Key words 1
1.2 Statement Type 1
2. Description 2
2.1 TODO (notes to keep the editor on track) 2
2.2 Questions 2
2.2.1 Delegation Questions 6
2.2.2 NV Questions 10
3. Protection 12
3.1 Introduction 12
3.2 Threat 13
3.3 Protection of functions 13
3.4 Protection of information 13
3.5 Side effects 14
3.6 Exceptions and clarifications 14
4. TPM Architecture 16
4.1 Interoperability 16
4.2 Components 16
4.2.1 Input and Output 17
4.2.2 Cryptographic Co-Processor 17
4.2.2.1 RSA Engine 18
4.2.2.2 Signature Operations 18
4.2.2.3 Symmetric Encryption Engine 18
4.2.2.4 Using Keys 19
4.2.3 Key Generation 20
4.2.3.1 Asymmetric – RSA 20
4.2.3.2 Nonce Creation 20
4.2.4 HMAC Engine 20
4.2.5 Random Number Generator 21
4.2.5.1 Entropy Source and Collector 22
4.2.5.2 State Register 22
4.2.5.3 Mixing Function 23
4.2.5.4 RNG Reset 23
4.2.6 SHA-1 Engine 24
4.2.7 Power Detection 24

4.2.8 Opt-In ... 24
4.2.9 Execution Engine ... 26
4.2.10 Non-Volatile Memory ... 26
4.3 Data Integrity Register (DIR) ... 26
4.4 Platform Configuration Register (PCR) ... 26
5. Endorsement Key Creation ... 29
5.1 Controlling Access to PRIVEK ... 30
5.2 Controlling Access to PUBEK ... 30
6. Attestation Identity Keys ... 31
7. TPM Ownership ... 32
7.1 Platform Ownership and Root of Trust for Storage ... 32
8. Authentication and Authorization Data ... 33
8.1 Dictionary Attack Considerations ... 34
9. TPM Operation ... 36
9.1 TPM Initialization & Operation State Flow ... 37
9.1.1 Initialization ... 37
9.2 Self-Test Modes ... 39
9.2.1 Operational Self-Test ... 41
9.3 Startup ... 45
9.4 Operational Mode ... 45
9.4.1 Enabling a TPM ... 46
9.4.2 Activating a TPM ... 48
9.4.3 Taking TPM Ownership ... 49
9.4.3.1 Enabling Ownership ... 50
9.4.4 Transitioning Between Operational States ... 51
9.5 Clearing the TPM ... 51
10. Physical Presence ... 53
11. Root of Trust for Reporting (RTR) ... 55
11.1 Platform Identity ... 55
11.2 RTR to Platform Binding ... 56
11.3 Platform Identity and Privacy Considerations ... 56
11.4 Attestation Identity Keys ... 56
11.4.1 AIK Creation ... 57
11.4.2 AIK Storage ... 58
12. Root of Trust for Storage (RTS) ... 59
12.1 Loading and Unloading Blobs ... 59
13. Transport Sessions and Authorization Protocols ... 60

13.1 Authorization Session Setup ... 63
13.2 Parameter Declarations for OIAP and OSAP Examples ... 64
13.2.1 Object-Independent Authorization Protocol (OIAP) ... 67
13.2.2 Object-Specific Authorization Protocol (OSAP) ... 71
13.3 Authorization Session Handles ... 75
13.4 Authorization-Data Insertion Protocol (ADIP) ... 76
13.5 AuthData Change Protocol (ADCP) ... 80
13.6 Asymmetric Authorization Change Protocol (AACP) ... 81
14. FIPS 140 Physical Protection ... 82
14.1 TPM Profile for FIPS Certification ... 82
15. Maintenance ... 83
15.1 Field Upgrade ... 84
16. Proof of Locality ... 86
17. Monotonic Counter ... 87
18. Transport Protection ... 90
18.1 Transport encryption and authorization ... 92
18.1.1 MGF1 parameters ... 94
18.1.2 HMAC calculation ... 94
18.1.3 Transport log creation ... 95
18.1.4 Additional Encryption Mechanisms ... 95
18.2 Transport Error Handling ... 95
18.3 Exclusive Transport Sessions ... 96
18.4 Transport Audit Handling ... 97
18.4.1 Auditing of wrapped commands ... 97
19. Audit Commands ... 99
19.1 Audit Monotonic Counter ... 101
20. Design Section on Time Stamping ... 102
20.1 Tick Components ... 102
20.2 Basic Tick Stamp ... 103
20.3 Associating a TCV with UTC ... 103
20.4 Additional Comments and Questions ... 105
21. Context Management ... 108
22. Eviction ... 110
23. Session pool ... 111
24. Initialization Operations ... 112
25. HMAC digest rules ... 114
26. Generic authorization session termination rules ... 115

27. PCR Grand Unification Theory ... 116
27.1 Validate Key for use ... 119
28. Non Volatile Storage ... 120
28.1 NV storage design principles ... 121
28.1.1 NV Storage use models ... 121
28.2 Use of NV storage during manufacturing ... 123
29. Delegation Model ... 124
29.1 Table Requirements ... 124
29.2 How this works ... 125
29.3 Family Table ... 127
29.4 Delegate Table ... 128
29.5 Delegation Administration Control ... 129
29.5.1 Control in Phase 1 ... 130
29.5.2 Control in Phase 2 ... 131
29.5.3 Control in Phase 3 ... 131
29.6 Family Verification ... 131
29.7 Use of commands for different states of TPM ... 133
29.8 Delegation Authorization Values ... 133
29.8.1 Using the authorization value ... 134
29.9 DSAP description ... 134
30. Physical Presence ... 138
30.1 Use of Physical Presence ... 138
31. TPM Internal Asymmetric Encryption ... 140
31.1.1 TPM_ES_RSAESOAEP_SHA1_MGF1 ... 140
31.1.2 TPM_ES_RSAESPKCSV15 ... 141
31.1.3 TPM_ES_SYM_CTR ... 141
31.1.4 TPM_ES_SYM_OFB ... 141
31.2 TPM Internal Digital Signatures ... 142
31.2.1 TPM_SS_RSASSAPKCS1v15_SHA1 ... 142
31.2.2 TPM_SS_RSASSAPKCS1v15_DER ... 143
31.2.3 TPM_SS_RSASSAPKCS1v15_INFO ... 143
31.2.4 Use of Signature Schemes ... 143
32. Key Usage Table ... 145
33. Direct Anonymous Attestation ... 147
33.1 TPM_DAA_JOIN ... 147
33.2 TPM_DAA_Sign ... 149
33.3 DAA Command summary ... 149

33.3.1 TPM setup 150
33.3.2 JOIN 150
33.3.3 SIGN 154
34. General Purpose IO 157
35. Redirection 158
36. Structure Versioning 159
37. Certified Migration Key Type 161
37.1 Certified Migration Requirements 161
37.2 Key Creation 162
37.3 Migrate CMK to a MA 162
37.4 Migrate CMK to a MSA 163
38. Revoke Trust 164
39. Mandatory and Optional Functional Blocks 166
40. 1.1a and 1.2 Differences 170

Copyright © TCG

1. Scope and Audience

The TPCA main specification is an industry specification that enables trust in computing platforms in general. The main specification is broken into parts to make the role of each document clear. A version of the specification (like 1.2) requires all parts to be a complete specification.

A TPM designer MUST be aware that for a complete definition of all requirements necessary to build a TPM, the designer MUST use the appropriate platform specific specification for all TPM requirements.

1.1 Key words

The key words “MUST,” “MUST NOT,” “REQUIRED,” “SHALL,” “SHALL NOT,” “SHOULD,” “SHOULD NOT,” “RECOMMENDED,” “MAY,” and “OPTIONAL” in the chapters 2-10 normative statements are to be interpreted as described in [RFC-2119].

1.2 Statement Type

Please note a very important distinction between different sections of text throughout this document. You will encounter two distinctive kinds of text: informative comment and normative statements. Because most of the text in this specification will be of the kind normative statements, the authors have informally defined it as the default and, as such, have specifically called out text of the kind informative comment They have done this by flagging the beginning and end of each informative comment and highlighting its text in gray. This means that unless text is specifically marked as of the kind informative comment, you can consider it of the kind normative statements.

For example:

Start of informative comment

This is the first paragraph of 1–n paragraphs containing text of the kind *informative comment* …

This is the second paragraph of text of the kind *informative comment* …

This is the nth paragraph of text of the kind *informative comment* …

To understand the TCG specification the user must read the specification. (This use of MUST does not require any action).

End of informative comment

This is the first paragraph of one or more paragraphs (and/or sections) containing the text of the kind normative statements …

To understand the TCG specification the user MUST read the specification. (This use of MUST indicates a keyword usage and requires an action).

2. Description

The design principles give the basic concepts of the TPM and generic information relative to TPM functionality.

A TPM designer MUST review and implement the information in the TPM Main specification (parts 1-4) and review the platform specific document for the intended platform. The platform specific document will contain normative statements that affect the design and implementation of a TPM.

A TPM designer MUST review and implement the requirements, including testing and evaluation, as set by the TCG Conformance Workgroup. The TPM MUST comply with the requirements and pass any evaluations set by the Conformance Workgroup. The TPM MAY undergo more stringent testing and evaluation.

The question section keeps track of questions throughout the development of the specification and hence can have information that is no longer current or moot. The purpose of the questions is to track the history of various decisions in the specification to allow those following behind to gain some insight into the committees thinking on various points.

2.1 TODO (notes to keep the editor on track)

2.2 Questions

How to version the flag structures?

I suggest that we simply put the version into the structure and pass it back in the structure. Add the version information into the persistent and volatile flag structures.

When using the encryption transport failures are easy to see. Also the watcher on the line can tell where the error occurred. If the failure occurs at the transport level the response is an error (small packet) and it is in the clear. If the error occurs during execution of the command then the response is a small encrypted packet. Should we expand the packet size or simply let this go through?

Not an issue.

Do we restrict the loading of a counter to once per TPM_Startup(Clear)?

Yes once a counter is set it must remain the same until the next successful startup.

Does the time stamp work as a change on the tag or as a wrapped command like the transport protection.

While possibly easier at the HW level the tag mechanism seems to be harder at the SW level as to what commands are sent to the TPM. The issue of how the SW presents the TS session to the SW writer is not an issue. This is due to the fact that however the session is presented to the SW writer the writer must take into account which commands are being time stamped and how to manage the log etc. So accepting a mechanism that is easy for the HW developer and having the SW manage the interface is a sufficient direction.

Copyright © TCG

When returning time information do we return the entire time structure or just the time and have the caller obtain all the information with a GetCap call?

All time returns will use the entire structure with all the details.

Do we want to return a real clock value or a value with some additional bits (like a monotonic value with a time value)?

Add a count value into the time structure.

Do we need NTP or is SNTP sufficient?

The TPM will not run the time protocol itself. What the TPM will do is accept a value from outside software and a hash of the protocols that produced the value. This allows the platform to use whatever they want to set the value from secure time to the local PC clock.

Can an owner destroy a TPM by issuing repeated CreateCounter commands?

A TPM may place a throttle on this command to avoid burn issues. It MUST not be possible to burn out the TPM counter under normal operating conditions. The CreateCounter command is limited to only once per successful TPM_Startup(ST_CLEAR).

This answer is now somewhat moot as the command to createcounter is now owner authorized. This allows the owner to decide when to authorize the counter creation. As there are only 4 counters available it is not an issue with having the owner continue to authorize counters.

What happens to a transport session (log etc.) on an S3?

Should these be the same as the authorization sessions? The saving of a transport session across S3 is not a security concern but is a memory concern. The TPM MUST clear the transport session on TPM_Startup(CLEAR) and MAY clear the session on TPM_Startup(any).

While you can't increment or create a new counter after startup can you read a counter other than the active one?

You may read other counters

When we audit a command that is not authorized should we hash the parameters and provide that as part of the audit event, currently they are set to null.

We should hash parameters of non-authorized commands

There is a fundamental problem with the encryption of commands in the transport and auditing. If we cover a command we have no way to audit, if we show the command then it isn't protected. Can we expose the command (ordinal) and not the parameters?

If the owner has requested that a function be audited then the execute transport return will include sufficient information to produce the audit entry.

How to set the time in the audit structure and tell the log what is going on.

The time in the audit structure is set to nulls except when audit occurs as part of a transport session. In that case the audit command is set from the time value in the TPM.

114 Is there a limit to the number of locality modifiers?

115 Yes, the TPM need only support a maximum of 4 modifiers. The definition of the
116 modifiers is always a platform specific issue.

117 How do we evict various resources?

118 There are numerous eviction routines in the current spec. We will deprecate the various
119 types and move to TPM_Flushxxx for all resource types.

120 Can you flush a saved context?

121 Yes, you must be able to invalidate saved contexts. This would be done by making sure
122 that the TPM could not load any saved context.

123 What is the value of maintaining the clock value when the time is not incrementing? Can
124 this be due to the fact that the time is now known to be at least after the indicated time?

125 Moot point now as we don't keep the clock value at

126 Should we change the current structures and add the tag?

127 TODO

128 Can we have a bank of bits (change bit locality) for each of the 4 levels of locality?

129 Now

130 How do we find out what sessions are active? Do we care?

131 I would say yes we care and we should use the same mechanism that we do for the keys.
132 A GetCap that will return the handles.

133 Can we limit the transport sessions to only one?

134 No, we should have as a minimum 2 sessions. One gets into deadlocks and such so the
135 minimum should be 2.

136 Does the TPM need to keep the audit structure or can it simply keep a hash?

137 The TPM just keeps the audit digest and no other information.

138 What happens to an OSAP session if the key associated with it is taken off chip with a
139 "SaveContext"? What happens if the key saveContext occurs after an OSAP auth context
140 that is already off chip? How do you later connect the key to the auth session (without
141 having to store all sorts of things on chip)? Are we really honestly convinced that we've
142 thought of all the possible ramifications of saving and restoring auth sessions? And is it
143 really true that all the things we say about a saved auth session do/should apply to a
144 saved key (which is to say is there really a single loadContext command and a single
145 context structure)?

146 Saved context a reliable indication of the linkage between the OSAP and the key. When
147 saving save auth then key, on load key then auth. Auth session checks for the key
148 and if not found fails.

149 Why is addNonce an output of 16.5 loadContext?

150 If it's wrong, it's a little late to find out now - why not have it as an input and have the
151 TPM return an error if the encrypted addNonce doesn't match the input? The thought
152 was that the nonce area might not be a nonce but was information that the caller

could put in. If they use it as a nonce fine, but they could also use it as a label or sequence number or … any value the caller wanted

Is there a memory endurance problem with contextNonceSession?

contextNonceSession does not have to be saved across S3 states so there is no endurance problem.

Is there a memory endurance problem with contextNonceKey?

contextNonceKey only changes on TPM_Startup(ST_Clear) so it's endurance is the same as a PCR.

The debate continues about restoring a resource's handle during TPM_LoadContext.

Debate ends by having the load context be informed of what the loaders opinion is about the handle. The requestor can indicate that it wishes the same handle and if the TPM can perform that task it does, if it cannot then the load fails.

Interesting attack is now available with the new audit close flag on get audit signed. Anyone with access to a signing key can close the audit log. The only requirement on the command is that the key be authorized. While there is no loss of information (as the attacker can always destroy the external log) does the closing of a log make things look different. This does enable a burn out attack. The ability to closeAudit enables a new DenialOfService attack.

Resolution: The TPM Owner owns the audit process, so the TPM Owner should have exclusive control over closeAudit. Hence the signing key used to closeAudit must be an AIK. Note that the owner can choose to give this AIK's AuthData value to the OS, so that the OS can automatically close an audit session during platform power down. But such operations are outside this specification.

Should we keep the E function in the tick counter?

From Graeme, I would prefer to see these calculations deleted. The calculation starts with one assertion and derives a contradictory assertion. Generally, there seems little value in trying to derive an equality relationship when nothing is known about the path to and from the Time Authority.

What is the difference between DIR_Quote and DirReadSigned?

Appears to be none so DIR_Quote deleted

The tickRate parameter associates tick with seconds and has no way to indicate that the rate is greater than one second. Is this OK?

Do we need to allow for tick rates that are slower than once per second. We report in nanoseconds.

The TPM MUST support a minimum of 2 authorization sessions. Where do we put this requirement in the spec?

Can we find a use for the DIR and BIT areas for locality 0?

They have no protections so in many ways they are just extra. We leave this as it is as locality 0 may mean something else on a platform other than a PC.

How do we send back the transport log information on each execute transport?

It is 64 byes in length and would make things very difficult to include on every command. Change wrappedaudit to be input params, add output parms and the caller has all information necessary to create the structure to add into the digest.

The transport log structure is a single structure used both for input and output with the only difference being the setting of ticks to 0 on input and a real value on output, do we need two structures.

I believe that a single structure is fine

For TPM_Startup(ST_Clear) I added that all keys would be flushed. Is this right?

Yes

Why have 2 auths for release transport signed? It is an easy attack to simply kill the session.

The reason is that an attacker can close the session and get a signature of the session log. We are currently not sure of the level of this attack but by having the creator of the session authorize the signing of the log it is completely avoided.

19.3 Action 3 (startup/state) doesn't reference the situation where there is no saved state. My presumption is that you can still run startup/clear, but maybe you have to do a hardware reset?

DWG I don't think so. This could be an attack and a way to get the wrong PCR values into the system. The BIOS is taking one path and may not set PCR values. Hence the response is to go into failed selftest mode.

What happens to a transport session if a command clears the TPM like revokeTrust

This is fine. The transport session is not complete but the session protected the information till the command that changed the TPM. It is impossible to get a log from the session or to sign the session but that is what the caller wanted.

2.2.1 Delegation Questions

Is loading the table by untrusted process ok? Does this cause a problem when the new table is loaded and permissions change?

Yes, the fill table can be done by any process. A TPM Owner wishing to validate the table can perform the operations necessary to gain assurance of the table entries.

Are the permissions for a table row sensitive?

Currently we believe not but there are some attack models that knowing the permissions makes the start of the attack easier. It does not make the success of the attack any easier. Example if I know that a single process is the only process in the table that has the CreateAIK capability then the attacker only attempts to break into the single process and not all others.

What software is in use to modify the table?

The table can be updated by any software or process given the capability to manage the table. Three likely sources of the software would be a BIOS process, an applet of a trusted process and a standalone self-booting (from CD-ROM) management application.

Who holds the TPM Owner password?

234 There is no change to the holding of the TPM Owner token. The permissions do allow the
235 creation of an application that sets the TPM Owner token to a random value and
236 then seals the value to the application.

237 How are these changes created such that there is minimal change to the current TPM?

238 This works by using the current authorization process and only making changes in the
239 authorization and not for each and every command.

240 What about S3 and other events?

241 Permissions, once granted, are non-volatile.

242 The permission bit to changeOwnerAuth (bit 11) gives rise to the functionality that the SW
243 that has this bit can control the TPM completely. This includes removing control from
244 the TPM Owner as the TPM Owner value will now be a random value only known to SW.
245 There are use models where this is good and bad, do we want this functionality?

246 Pros and cons of physical enable table when TPM Owner is present – Pro physically present
247 user can make SW play fair. Con – physically present user can override the desires of a
248 TPM Owner.

249 Do we need to reset TPM_PERMISSION_KEY at some time?

250 We know that the key is NOT reset on TPM_ClearOwner.

251 What is the meaning of using permission table in an OIAP and OSAP mode?

252 Delegate table can be used in either OIAP or OSAP mode.

253 Can you grant permissions without assigning the permissions to a specific process?

254 Yes, do a SetRow with a PCR_SELECTION of null and the permissions are available to
255 any process.

256 Do we need a ClearTableOwner?

257 I would assert that we do not need this command. The TPM Owner can perform SetRow
258 with NULLS four times and creates the exact same thing. Not having this command
259 lowers the number of ordinals the TPM is required to support.

260 There are some issues with the currently defined behavior of familyID and the
261 verificationCount.

262 Talked to David for 30 mins. We decided that maxFamilyID is set to zero at
263 manufacture, and incremented for every FamTable_SetRow

264 It is the responsibility of DelTable_SetRow to set the appropriate familyID

265 DelTable_SetRow fails if the provided familyID is not active and present somewhere in
266 the FamTable

267 FillTable works differently. It effectively resets the family table (invalidating all active
268 rows) and sets up as many rows as are needed based on the number of families
269 specified in FillTable

270 This still needs a bit of work. Presumably the caller of FillTable uses a “fake” familyID,
271 and this is changed to the actual familyID when the fill happens

272 There are some issues with the verificationCount.

273 Uber-issue. If none of the rows in the table are allowed to create other rows and export
274 them, then the "sign" of the table is meaningful

275 If one of the rows is allowed to create and export new rows, is there any real meaning to
276 "the current set of exported rows?" (i.e. SW can just up and make new rows).

277 Should section 4.4, TPM_DelTable_ClearTable), section 4.5 (TPM_DelTable_SetEnable), and
278 section 4.7 (TPM_DelTable_Set_Admin) all say "there must be UNAMBIGUOUS evidence
279 of the presence of physical access…" Is this okay?

280 Answer: No, group agreed to change UNAMBIGUOUS to BEST EFFORT in all three
281 sections.

282 Is FamilyID a sensitive value?

283 If so, why? Agreement: FamilyID is not a sensitive value.

284 Should TPM_TakeOwnership be included in permissions bits (see bit 12 in section 3.1)?

285 Enables a better administrative monitor and may enable user to take ownership easier.
286 Agreement leave it in and change informative comments to reflect the reasons.

287 [From the TPM_DelTable_SetRow command informative comments]: Note that there are two
288 types of rights: family rights (you can either edit your family's rows or grab new rows)
289 and administrative rights.

290 This is really just an editor's note, not a question to be resolved.

291 [From the TPM_DelTable_ExportRow command informational comments]:

292 Does not effect content of exported row left behind in the table;

293 Valid for all rows in the table;

294 Does not need to be OwnerAuth'd;

295 Family Rights are that family can only export a row from rows 0-3 if row belongs to the
296 family, but rows 4 and upwards can be exported by any Trusted Process, without any
297 family checking being done. This is really just an editor's note, not a question to be
298 resolved.

299 When a Family Table row is set, the verificationCount is set to 1, make sure that is
300 consistently used in all other command actions.

301 Done.

302 SetEnable and SetEnableOwner enable and disable all rows in a table, not just the rows
303 belong to the family of the process that used the SetEnable and/or SetEnableOwner
304 commands. This is also true for SetAdmin and SetAdminOwner. Can anybody come up
305 with a use scenario where that causes any problems?

306 In command actions where the TPM must walk the delegation table looking for a
307 configuration that matches the command input parameters (PCRinfo and/or
308 authValues) and there are rows in the table with duplicate values, what does the TPM
309 do? Is there any reason not to use the rule "the TPM starts walking the table starting
310 with the first row and use the first row it finds with matching values"?

311 Answer to this question may mean change to pseudo code in section 2.3, Using the
312 AuthData Value, which currently shows the TPM walking the delegation table,
313 starting with the first row, and using the first row it finds with matching values.

314 What familyID value signals a family table row that is not in use/contains invalid values?

315 To get consistency in all the command Actions that use this, that FamilyID value has
316 been edited in all places to be NULL, instead of 0. Yes, FamilyID value of NULL
317 signals a family table row that is not in use or contains invalid values.

318 From section 2.4, Delegate Table Fill and Enablement: "The changing of a TPM Owner does
319 not automatically clear the delegate table. Changing a TPM Owner does disable all
320 current delegations, including exported rows, and requires the new TPM Owner to re-
321 enable the delegations in the table. The table entry values like trusted process
322 identification and delegations to that process are not effected by a change in owner. THE
323 AUTHDATA VALUES DO NOT SURVIVE THE OWNERSHIP CHANGE." Question: If this is
324 true, no delegations work after a change of owner. How does the new owner set new
325 AuthData values?

326 The simple way of handling this is to get AdminMonitor to own backing up delegations at
327 first owner install and then be run by new owner, and AdminMonitor uses FillTable,
328 to handle "Owner migration." Or, for another use option, is for second owner to pick-
329 up PCR-ID's and delegations bits from previous owner – what is the most straight-
330 forward way to do this?

331 In section 3.1 (Delegate Definitions bit map table), several commands that do not require
332 owner authorization are in the table and can be delegated: TPM_SetTempDeactivated (bit
333 15), TPM_ReadPubek (bit 7), and TPM_LoadManuMaintPub (bit 3), Why?

334 In section 3.3 it is stated, "The Family ID resets to NULL on each change of TPM Owner."
335 This invalidates all delegations. Is this what we want?

336 You don't have to blow away FamilyID to blow away the blobs, because key is gone. So
337 this is not required – can eliminate these actions.

338 In section 3.12, why is TPM_DELEGATE_LABEL included in the table?

339 In section 4.2 (TPM_DelTable_FillTable), is it okay to delete requirement that delegate table
340 be empty? Also, in Action 14, now that we have both persistent and volatile tableAdmin
341 flags, should this command set volatile tableAdmin flag to FALSE upon completion?

342 The delegate table does not need to be empty to use the TPM_DelTable_FillTable
343 command, Also, a paragraph has been added to Informative comment for
344 TPM_DelTable_FillTable that points out usefulness of immediately following
345 TPM_DelTable_FillTable with TPM_Delegate_TempSetAdmin, to stop table
346 administration in the current boot cycle.

347 In section 4.15 (TPM_FamTable_IncrementCount), why does this command require
348 TPMOwner authorization, as currently documented in section 4.15?

349 IncrementCount is gated by tableAdmin, which seems sufficient, and use of ownerauth
350 makes it difficult to automatically verify a table using a CDROM.

351 In section 4.3 (TPM_DelTable_FillTableOwner), in the Action 3d, use OTP[80] = MFG(x1) in
352 place of oneTimePad[n] = SHA1(x1 || seed[n]))?,

353 yes.

354 In section 4.9 (TPM_DelTable_SetRow), is invalidateRow input parameter really needed?

355 It is only used in action 5. Couldn't action 5 simply read "Set N1 -> familyID = NULL"?

There is no easy way to generate a blob that can be used to delegate migration authority for a user key.

This is because the TPM does not store the migration authority on the chip as the migration command involves an encrypted key, not a loaded one. One could invent a 'CreateMigrationDelegationBlob' that took the encrypted key as input and generated the encrypted delegation blob as output, but it would not be pretty. Sorry Dave.

If a delegate row in NV memory (nominally 4 rows) is to refer to a user key (instead of owner auth), then it needs to include a hash of the public key. It could be that the NV table is restricted to owner auth delegations, this would save 80 bytes of NV store and also simplify the LoadBlob command.

Maybe would simplify other things. I would definitely NOT permit user keys in the table to be run with the legacy OSAP and OIAP ordinals.

A few more GetCapability values are also required, the usual constants that we discussed and also the two readTable caps.

TBD Verify that Delegate Table Management commands (see section 2.8) cover all the functionality of obsolete or updated commands.

Redefine bits 16 and above in Delegation Definitions table (section 3.1). In particular, can new command set (with TPM_FAMILY_OPERATION options as defined in section 3.20) be delegated individually and appropriately. Also, how many user key authorized commands will be delegated?

Is new TPM_FAMILY_FLAGS field of family table (defined in section 3.5) sensitive data?

DSAP informative comment needs to be completed (section 4.1). In particular, does the statement "The DSAP command works like OSAP except it takes an encrypted blob – an encrypted delegate table row -- as input" sufficient? Or do some particular differences between DSAP and OSAP have to be pointed out in this informative comment??

The TPM_Delegate_LoadBlob[Owner] commands cannot be used to load key delegation blobs into the TPM. Is another ordinal required to do that?

Is it okay for TPM_Delegate_LoadBlob[Owner] commands to ignore enable/disable use/admin flags in family table rows?

Is it wise to delegate TPM_DelTable_ConvertBlob command (defined in section 4.11)? Does current definition of this command support section 2.7 scenarios?

Is there a privacy problem with DelTable_ReadRow since the contents may not be identical from TPM to TPM?

Are DSAP sessions being pooled with the other sessions? if so, can one save\load them by context functions? if not, then there should be a restriction in saveContext.

DSAP are "normal" authorization sessions and would save/load with OIAP and OSAP sessions

2.2.2 NV Questions

You would set this by using a new ordinal that is unauthorized and only turns the flag on to lock everything. Yet another ordinal? Do we need it? Is this an important functionality for the uses we see?

397 Yes this allows us to have "close" to writeonce functionality. What the functionality
398 would be is that the RTM would assure that the proper information is present in the
399 TPM and then "lock" the area. One could create this functionality by having the RTM
400 change the authorization each time but then you would need to eat more NV store so
401 save the sealed AuthData value. I think that is easier to have an ordinal than eat the
402 NV space and require a much more complex programming model.

403 Is it OK to have an element partially written?

404 Given that we have chunks there has to be a mechanism to allow partial writes.

405 If an element is partially written, how does a caller know that more needs to be written?

406 I would say the use model that provides the ability to write – read, in a loop is just not
407 supported. Get it all written and then do the read.

408 Usage of the lock bit: as you wrote, the RTM would assure that the proper information is
409 present in the TPM and then "lock" the area. so why in action #4 we should also check
410 bWritten when the lock bit is set? should be as action #3b of TPM_NV_DefineSpace, if
411 lock is set - return error

412 [Grawrock, David] Not quite, the use model I was trying to create was the one where the
413 TPM was locked and the user was attempting to add a new area. If the locked bit
414 doesn't allow for writing once to a new area, one must reboot to perform the write
415 and also tell the RTM what the value to write must be. So this allows the creator of
416 an area to write it once and then it flows with the locked bit.

417 Can you delete a NV value with only physical presence?

418 [Grawrock, David] You can't delete with physical presence, you must use owner
419 authorization. This I think is a reasonable restriction to avoid burn problems.

420 Why is there no check on the writes for a TPM Owner?

421 The check for an owner occurred during the TPM_NV_DefineSpace. It is imperative that
422 the TPM_NV_DefineSpace set in place the appropriate restrictions to limit the
423 potential for attacks on the NV storage area.

424 Description of maxNVBufSize is confusing to me. Why is this value related to the input size?
425 And since there is no longer any 'written' bits, why is there a maximum area size at all?

426 [Grawrock, David] This is a fixed size and set by the TPM manufacturer. I would see
427 values like the input buffer, transport sessions etc all coming up with the max size
428 the TPM can handle. This does NOT indicate what is available on the TPM right now.
429 The TPM could have 4k of space but max size would be 782 and would always report
430 that number. If the available space fell to 20 bytes this value would still be 782.

431 If the storage area is an opaque area to the TPM (as described), then how does the TPM
432 know what PCR registers have been used to seal a blob?

433 The VALUES of the area are opaque, the attributes to control access are not. So if the
434 attributes indicate that PCR restrictions are in place the TPM keeps those PCR values
435 as part of the index attributes. This in reality seals the value as there is no need for
436 tpmProof since the value never leaves the TPM.

3. Protection

3.1 Introduction

Start of informative comment

The Protection Profile in the Conformance part of the specification defines the threats that are resisted by a platform. This section, "Protection," describes the properties of selected capabilities and selected data locations within a TPM that has a Protection Profile and has not been modified by physical means.

This section introduces the concept of protected capabilities and the concept of shielded locations for data. The ordinal set defined in part II and III is the set of protected capabilities. The data structures in part II define the shielded locations.

- A protected capability is one whose correct operation is necessary in order for the operation of the TCG Subsystem to be trusted.
- A shielded location is an area where data is protected against interference and prying, independent of its form.

This specification uses the concept of protected capabilities so as to distinguish platform capabilities that must be trustworthy. Trust in the TPM depends critically on the protected capabilities. Platform capabilities that are not protected capabilities must (of course) work properly if the TCG Subsystem is to function properly.

This specification uses the concept of shielded locations, rather than the concept of "shielded data." While the concept of shielded data is intuitive, it is extraordinarily difficult to define because of the imprecise meaning of the word "data." For example, consider data that is produced in a safe location and then moved into ordinary storage. It is the same data in both locations, but in one it is shielded data and in the other it is not. Also, data may not always exist in the same form. For example, it may exist as vulnerable plaintext, but also may sometimes be transformed into a logically protected form. This data continues to exist, but doesn't always need to be shielded data - the vulnerable form needs to be shielded data, but the logically protected form does not. If a specific form of data requires protection against interference or prying, it is therefore necessary to say "if the data-D exists, it must exist only in a shielded location." A more concise expression is "the data-D must be extant only in a shielded location."

Hence, if trust in the TCG Subsystem depends critically on access to certain data, that data should be extant only in a shielded location and accessible only to protected capabilities. When not in use, such data could be erased after conversion (using a protected capability) into another data structure. Unless the other data structure was defined as one that must be held in a shielded location, it need not be held in a shielded location.

End of informative comment

1. The data structures described in part II of the TPM specifications MUST NOT be instantiated in a TPM, except as data in TPM-shielded-locations.
2. The ordinal set defined in part II and III of the TPM specifications MUST NOT be instantiated in a TPM, except as TPM-protected-capabilities.
3. Functions MUST NOT be instantiated in a TPM as TPM-protected-capabilities if they do not appear in the ordinal set defined in part II and III of the TPM specifications.

3.2 Threat

Start of informative comment

This section, "Threat," defines the scope of the threats that must be considered when considering whether a platform facilitates subversion of capabilities and data in a platform.

The design and implementation of a platform determines the extent to which the platform facilitates subversion of capabilities and data within that platform. It is necessary to define the attacks that must be resisted by TPM-shielded locations and TPM-protected capabilities in that platform.

The TCG specifications define all attacks that are resisted by the TPM. These attacks must be considered when determining whether the integrity of TPM-protected capabilities and data in TPM-shielded locations can be damaged. These attacks must be considered when determining whether there is a backdoor method of obtaining access to TPM-protected capabilities and data in TPM-shielded locations. These attacks must be considered when determining whether TPM-protected capabilities have undesirable side effects.

End of informative comment

1. For the purposes of the "Protection" section of the specification, the threats that MUST be considered when determining whether the TPM facilitates subversion of TPM-protected-capabilities or data in TPM-shielded-locations SHALL include
   a. The methods inherent in physical attacks that fail if the TPM complies with the "physical protection" requirements specified by TCG
   b. All methods that require execution of instructions in a computing engine in the platform

3.3 Protection of functions

Start of informative comment

A TPM-protected-capability must be used to modify TPM-protected capabilities. Other methods must not be allowed to modify TPM-protected capabilities. Otherwise, the integrity of TPM-protected capabilities is unknown.

End of informative comment

1. A TPM SHALL NOT facilitate the alteration of TPM-protected-capabilities, except by TPM-protected capabilities.

3.4 Protection of information

Start of informative comment

TPM-protected capabilities must provide the only means from outside the TPM to access information represented by data in TPM-shielded-locations. Otherwise, a rogue can reveal data in TPM-shielded-locations, or create a derivative of data from TPM-shielded-locations (in a way that maintains some or all of the information content of the data) and reveal the derivative.

End of informative comment

1. A TPM SHALL NOT export data that is dependent upon data structures described in part II of the TPM specifications, other than via a TPM-Protected-Capability.

3.5 Side effects

Start of informative comment

An implementation of a TPM-protected capability must not disclose the contents of TPM-shielded locations. The only exceptions are when such disclosure is inherent in the definition of the capability or in the methods used by the capability. For example, a capability might be designed specifically to reveal hidden data or might use cryptography and hence always be vulnerable to cryptanalysis. In such cases, some disclosure or risk of disclosure is inherent and cannot be avoided. Other forms of disclosure (by side effects, for example) must always be avoided.

End of informative comment

1. The implementation of a TPM-protected-capability in a TPM SHALL NOT facilitate the disclosure or the exposure of information represented by data in TPM-shielded–locations, except by means unavoidably inherent in the TPM definition.

3.6 Exceptions and clarifications

Start of informative comment

These exceptions to the blanket statements in the generic "protection" requirements (above) are fully compatible with the intended effect of those statements. These exceptions affect TCG-data that is available as plain-text outside the TPM and TCG-data that can be used without violating security or privacy. These exceptions are valuable because they approve use of TPM resources by vendor-specific commands in particular circumstances.

These clarifications to the blanket statements of the generic "protection" requirements (above) do not materially change the effect of those statements, but serve to approve specific legitimate interpretations of the requirements.

End of informative comment

1. A Shielded Location is a place (memory, register, etc.) where data is protected against interference and exposure, independent of its form
2. A TPM-Protected-Capability is an operation defined in and restricted to those identified in part II and III of the TPM specifications.
3. A vendor specific command or capability MAY use the standard TCG owner/operator authorization mechanism
4. A vendor specific command or capability MAY utilize a TPM_PUBKEY structure stored on the TPM so long as the usage of that TPM_PUBKEY structure is authorized using the standard TCG authorization mechanism.
5. A vendor specific command or capability MAY use a sequence of standard TCG commands. The command MUST propagate the locality used for the call to the used TCG commands or capabilities, or set locality to 0.
6. A vendor specific command or capability that takes advantage of exceptions and clarifications to the "protection" requirements MUST be defined as part of the security

Copyright © TCG

557 target of the TPM. Such a vendor specific command or capability MUST be evaluated to
558 meet the Platform Specific TPM and System Security Targets.

559 7. If a TPM employs vendor-specific cipher-text that is protected against subversion to the
560 same or greater extent as internal TPM-resources stored outside the TPM with TCG-
561 defined methods, that vendor-specific cipher-text does not necessarily require protection
562 from physical attack. If a TPM location stores only vendor-specific cipher-text that does
563 not require protection from physical attack, that location can be ignored when
564 determining whether the TPM complies with the "physical protection" requirements
565 specified by TCG.

4. TPM Architecture

4.1 Interoperability

Start of informative comment

The TPM must support a minimum set of algorithms and operations to meet TCG specifications.

Algorithms

RSA, SHA-1, HMAC

The algorithms and protocols are the minimum that the TPM must support. Additional algorithms and protocols may be available to the TPM. All algorithms and protocols available in the TPM must be included in the TPM and platform credential.

The reason to specify these algorithms is two fold. The first is to know and understand the security properties of selected algorithms; identify appropriate key sizes and ensure appropriate use in protocols. The second reason is to define a base level of algorithms for interoperability.

End of informative comment

4.2 Components

Start of informative comment

The following is a block diagram Figure 4:a shows the major components of a TPM.

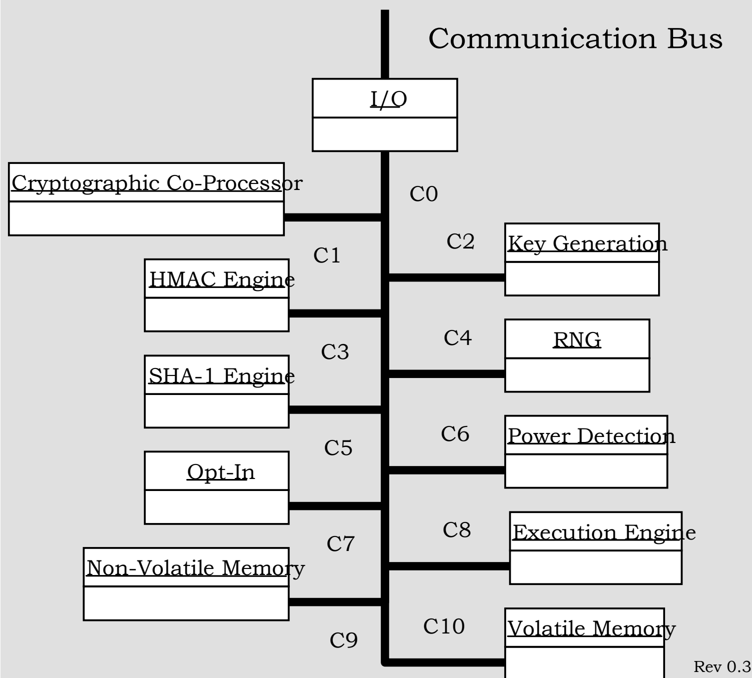


Figure 4:a - TPM Component Architecture

End of informative comment

587 ### 4.2.1 Input and Output

588 **Start of informative comment**

589 The I/O component, Figure 4:a C0, manages information flow over the communications
590 bus. It performs protocol encoding/decoding suitable for communication over external and
591 internal buses. It routes messages to appropriate components. The I/O component enforces
592 access policies associated with the Opt-In component as well as other TPM functions
593 requiring access control.

594 The main specification does not require a specific I/O bus. Issues around a particular I/O
595 bus are the purview of a platform specific specification.

596 **End of informative comment**

597 1. The number of incoming operand parameter bytes must exactly match the
598 requirements of the command ordinal. If the command contains more or fewer bytes
599 than required, the TPM MUST return TPM_BAD_PARAMETER.

600 ### 4.2.2 Cryptographic Co-Processor

601 **Start of informative comment**

602 The cryptographic co-processor, Figure 4:a C1, implements cryptographic operations within
603 the TPM. The TPM employs conventional cryptographic operations in conventional ways.
604 Those operations include the following:

605 Asymmetric key generation (RSA)

606 Asymmetric encryption/decryption (RSA)

607 Hashing (SHA-1)

608 Random number generation (RNG)

609 The TPM uses these capabilities to perform generation of random data, generation of
610 asymmetric keys, signing and confidentiality of stored data.

611 The TPM may symmetric encryption for internal TPM use but does not expose any
612 symmetric algorithm functions to general users of the TPM.

613 The TPM may implement additional asymmetric algorithms. TPM devices that implement
614 different algorithms may have different algorithms perform the signing and wrapping.

615 If the TPM uses RSA with the required key length (2048 bits for storage keys), the output of
616 all commands for key or data blob generation (e.g., TPM_CreateWrapKey, TPM_Seal,
617 TPM_Sealx, TPM_MakeIdentity) consists of only one block. However, if the TPM uses other
618 asymmetric algorithms that result in more than one output block for these commands, the
619 integrity of the blobs must be protected by the TPM (by means of appropriate chaining
620 mechanisms).

621 **End of informative comment**

622 1. The TPM MAY implement other asymmetric algorithms such as DSA or elliptic curve.
623 a. These algorithms may be in use for wrapping, signatures and other operations. There
624 is no guarantee that these keys can migrate to other TPM devices or that other TPM
625 devices will accept signatures from these additional algorithms.

626 b. If the output key or data blob generated with a storage key consists of more than one
627 block, the TPM MUST protect the integrity of the blob by means of appropriate
628 chaining mechanisms.

629 2. All storage keys MUST be of strength equivalent to a 2048 bits RSA key or greater. The
630 TPM SHALL NOT load a storage key whose strength less than that of a 2048 bits RSA
631 key.

632 3. All AIK MUST be of strength equivalent to a 2048 bits RSA key, or greater.

633 #### 4.2.2.1 RSA Engine

634 **Start of informative comment**

635 The RSA asymmetric algorithm is used for digital signatures and for encryption.

636 For RSA keys the PKCS #1 standard provides the implementation details for digital
637 signature, encryption and data formats.

638 There is no requirement concerning how the RSA algorithm is to be implemented. TPM
639 manufacturers may use Chinese Remainder Theorem (CRT) implementations or any other
640 method. Designers should review P1363 for guidance on RSA implementations.

641 **End of informative comment**

642 1. The TPM MUST support RSA.

643 2. The TPM MUST use the RSA algorithm for encryption and digital signatures.

644 3. The TPM MUST support key sizes of 512, 1024, and 2048 bits. The TPM MAY support
645 other key sizes.

646 a. The minimum RECOMMENDED key size is 2048 bits.

647 4. The RSA public exponent MUST be e, where $e = 2^{16}+1$.

648 5. TPM devices that use CRT as the RSA implementation MUST provide protection and
649 detection of failures during the CRT process to avoid attacks on the private key.

650 #### 4.2.2.2 Signature Operations

651 **Start of informative comment**

652 The TPM performs signatures on both internal items and on requested external blobs. The
653 rules for signatures apply to both operations.

654 **End of informative comment**

655 1. The TPM MUST use the RSA algorithm for signature operations where signed data is
656 verified by entities other than the TPM that performed the sign operation.

657 2. The TPM MAY use other asymmetric algorithms for signatures; however, there is no
658 requirement that other TPM devices either accept or verify those signatures.

659 3. The TPM MUST use P1363 for the format and design of the signature output.

660 #### 4.2.2.3 Symmetric Encryption Engine

661 **Start of informative comment**

662 The TPM uses symmetric encryption to encrypt authentication information, provide
663 confidentiality in transport sessions and provide internal encryption of blobs stored off the
664 TPM.

665 For authentication and transport sessions, the mandatory mechanism is a Vernam one-
666 time-pad with XOR. The mechanism to generate the one-time-pad is MGF1 and the nonces
667 from the session protocol. When encrypting authorization data, the authorization data and
668 the nonces are the same size, 20 bytes, so a direct XOR is possible.

669 For transport sessions the size of data is larger than the nonces so there needs to be a
670 mechanism to expand the entropy to the size of the data. The mechanism to expand the
671 entropy is the MGF1 function from PKCS#1. This function provides a known mechanism
672 that does not lower the entropy of the nonces.

673 AES may be supported as an alternate symmetric key encryption algorithm.

674 Internal protection of information can use any symmetric algorithm that the TPM designer
675 feels provides the proper level of protection.

676 The TPM does not expose any of the symmetric operations for general message encryption.

677 **End of informative comment**

678 4.2.2.4 Using Keys

679 **Start of Informative comments:**

680 Keys can be symmetric or asymmetric.

681 As the TPM does not have an exposed symmetric algorithm, the TPM is only a generator,
682 storage device and protector of symmetric keys. Generation of the symmetric key would use
683 the TPM RNG. Storage and protection would be provided by the BIND and SEAL capabilities
684 of the TPM. If the caller wants to ensure that the release of a symmetric key is not exposed
685 after UNBIND/UNSEAL on delivery to the caller, the caller should use a transport session
686 with confidentiality set.

687 For asymmetric algorithms, the TPM generates and operates on RSA keys. The keys can be
688 held only by the TPM or in conjunction with the caller of the TPM. If the private portion of a
689 key is in use outside of the TPM it is the responsibility of the caller and user of that key to
690 ensure the protections of the key.

691 The TPM has provisions to indicate if a key is held exclusively for the TPM or can be shared
692 with entities off of the TPM.

693 **End of informative comments.**

694 1. A secret key is a key that is a private asymmetric key or a symmetric key.

695 2. Data SHOULD NOT be used as a secret key by a TCG protected capability unless that
696 data has been extant only in a shielded location.

697 3. A key generated by a TCG protected capability SHALL NOT be used as a secret key
698 unless that key has been extant only in a shielded location.

699 4. A secret key obtained by a TCG protected capability from a Protected Storage blob
700 SHALL be extant only in a shielded location.

701 ### 4.2.3 Key Generation

702 **Start of informative comment**

703 The Key Generation component, Figure 4:a C2, creates RSA key pairs and symmetric keys.
704 TCG places no minimum requirements on key generation times for asymmetric or
705 symmetric keys.

706 **End of informative comment**

707 #### 4.2.3.1 Asymmetric – RSA

708 The TPM MUST generate asymmetric key pairs. The generate function is a protected
709 capability and the private key is held in a shielded location. The implementation of the
710 generate function MUST be in accordance with P1363.

711 The prime-number testing for the RSA algorithm MUST use the definitions of P1363. If
712 additional asymmetric algorithms are available, they MUST use the definitions from P1363
713 for the underlying basis of the asymmetric key (for example, elliptic curve fitting).

714 #### 4.2.3.2 Nonce Creation

715 The creation of all nonce values MUST use the next n bits from the TPM RNG.

716 ### 4.2.4 HMAC Engine

717 **Start of informative comment**

718 The HMAC engine, Figure 4:a C3, provides two pieces of information to the TPM: proof of
719 knowledge of the AuthData and proof that the request arriving is authorized and has no
720 modifications made to the command in transit.

721 The HMAC definition is for the HMAC calculation only. It does not specify the order or
722 mechanism that transports the data from caller to actual TPM.

723 The creation of the HMAC is order dependent. Each command has specific items that are
724 portions of the HMAC calculation. The actual calculation starts with the definition from
725 RFC 2104.

726 RFC 2104 requires the selection of two parameters to properly define the HMAC in use.
727 These values are the key length and the block size. This specification will use a key length
728 of 20 bytes and a block size of 64 bytes. These values are known in the RFC as K for the key
729 length and B as the block size.

730 The basic construct is

731 H(K XOR opad, H(K XOR ipad, text))

732 where

733 H = the SHA1 hash operation

734 K = the key or the AuthData

735 XOR = the xor operation

736 opad = the byte 0x5C repeated B times

737 B = the block length

Copyright © TCG

738 ipad = the byte 0x36 repeated B times

739 text = the message information and any parameters from the command

740 **End of informative comment**

741 The TPM MUST support the calculation of an HMAC according to RFC 2104.

742 The size of the key (K in RFC 2104) MUST be 20 bytes. The block size (B in RFC 2104)
743 MUST be 64 bytes.

744 The order of the parameters is critical to the TPM's ability to recreate the HMAC. Not all of
745 the fields are sent on the wire for each command for instance only one of the nonce values
746 travels on the wire. Each command interface definition indicates what parameters are
747 involved in the HMAC calculation.

748
4.2.5 Random Number Generator

749 **Start of informative comment**

750 The Random Number Generator (RNG) component, Figure 6:a C4 is the source of
751 randomness in the TPM. The TPM uses these random values for nonces, key generation,
752 and randomness in signatures.

753 The RNG consists of a state-machine that accepts and mixes unpredictable data and a post-
754 processor that has a one-way function (e.g. SHA-1). The idea behind the design is that a
755 TPM can be good source of randomness without having to require a genuine source of
756 hardware entropy.

757 The state-machine can have a non-volatile state initialized with unpredictable random data
758 during TPM manufacturing before delivery of the TPM to the customers. The state-machine
759 can accept, at any time, further (unpredictable) data, or entropy, to salt the random
760 number. Such data comes from hardware or software sources – for example; from thermal
761 noise, or by monitoring random keyboard strokes or mouse movements. The RNG requires a
762 reseeding after each reset of the TPM. A true hardware source of entropy is likely to supply
763 entropy at a higher baud rate than a software source.

764 When adding entropy to the state-machine, the process must ensure that after the addition,
765 no outside source can gain any visibility into the new state of the state-machine. Neither
766 the Owner of the TPM nor the manufacturer of the TPM can deduce the state of the state-
767 machine after shipment of the TPM. The RNG post-processor condenses the output of the
768 state-machine into data that has sufficient and uniform entropy. The one-way function
769 should use more bits of input data than it produces as output.

770 Our definition of the RNG allows implementation of a Pseudo Random Number Generator
771 (PRNG) algorithm. However, on devices where a hardware source of entropy is available, a
772 PRNG need not be implemented. This specification refers to both RNG and PRNG
773 implementations as the RNG mechanism. There is no need to distinguish between the two
774 at the TCG specification level.

775 The TPM should be able to provide 32 bytes of randomness on each call. Larger requests
776 may fail with not enough randomness being available.

777 **End of informative comment**

778 1. The RNG for the TPM will consist of the following components:

779 a. Entropy source and collector
780 b. State register
781 c. Mixing function
782 2. The RNG capability is a TPM-protected capability with no access control.
783 3. The RNG output may or may not be shielded data. When the data is for internal use by
784 the TPM (e.g., generation of tpmProof or an asymmetric key), the data MUST be held in a
785 shielded location. The RNG output for internal use MUST not be known outside the
786 TPM. In particular, it MUST not be known by the TPM manufacturer. When the data is
787 for use by the TSS or another external caller, the data is not shielded.

788
4.2.5.1 Entropy Source and Collector

789 **Start of informative comment**

790 The entropy source is the process or processes that provide entropy. These types of sources
791 could include noise, clock variations, air movement, and other types of events.

792 The entropy collector is the process that collects the entropy, removes bias, and smoothes
793 the output. The collector differs from the mixing function in that the collector may have
794 special code to handle any bias or skewing of the raw entropy data. For instance, if the
795 entropy source has a bias of creating 60 percent 1s and only 40 percent 0s, then the
796 collector design takes that bias into account before sending the information to the state
797 register.

798 **End of informative comment**

799 1. The entropy source MUST provide entropy to the state register in a manner that provides
800 entropy that is not visible to an outside process.
801 a. For compliance purposes, the entropy source MAY be outside of the TPM; however,
802 attention MUST be paid to the reporting mechanism.
803 2. The entropy source MUST provide the information only to the state register.
804 a. The entropy source may provide information that has a bias, so the entropy collector
805 must remove the bias before updating the state register. The bias removal could use
806 the mixing function or a function specifically designed to handle the bias of the
807 entropy source.
808 b. The entropy source can be a single device (such as hardware noise) or a combination
809 of events (such as disk timings). It is the responsibility of the entropy collector to
810 update the state register whenever the collector has additional entropy.

811
4.2.5.2 State Register

812 **Start of informative comment**

813 The state register implementation may use two registers: a non-volatile register rngState
814 and a volatile register. The TPM loads the volatile register from the non-volatile register on
815 startup. Each subsequent change to the state register from either the entropy source or the
816 mixing function affects the volatile state register. The TPM saves the current value of the
817 volatile state register to the non-volatile register on TPM power-down. The TPM may update
818 the non-volatile register at any other time. The reasons for using two registers are:

Copyright © TCG

819 To handle an implementation in which the non-volatile register is in a flash device;

820 To avoid overuse of the flash, as the number of writes to a flash device are limited.

821 **End of informative comment**

822 1. The state register is in a TPM shielded-location.

823 a. The state register MUST be non-volatile.

824 b. The update function to the state register is a TPM protected-capability.

825 c. The primary input to the update function SHOULD be the entropy collector.

826 2. If the current value of the state register is unknown, calls made to the update function
827 with known data MUST NOT result in the state register ending up in a state that an
828 attacker could know.

829 a. This requirement implies that the addition of known data MUST NOT result in a
830 decrease in the entropy of the state register.

831 3. The TPM MUST NOT export the state register.

832 #### 4.2.5.3 Mixing Function

833 **Start of informative comment**

834 The mixing function takes the state register and produces output. The mixing function is a
835 TPM protected-capability. The mixing function takes the value from a state register and
836 creates the RNG output. If the entropy source has a bias, then the collector takes that bias
837 into account before sending the information to the state register.

838 **End of informative comment**

839 1. Each use of the mixing function MUST affect the state register.

840 a. This requirement is to affect the volatile register and does not need to affect the non-
841 volatile state register.

842 #### 4.2.5.4 RNG Reset

843 **Start of informative comment**

844 The resetting of the RNG occurs at least in response to a loss of power to the device.

845 These tests prove only that the RNG is still operating properly; they do not prove how much
846 entropy is in the state register. This is why the self-test checks only after the load of
847 previous state and may occur before the addition of more entropy.

848 **End of informative comment**

849 1. The RNG MUST NOT output any bits after a system reset until the following occurs:

850 a. The entropy collector performs an update on the state register. This does not include
851 the adding of the previous state but requires at least one bit of entropy.

852 b. The mixing function performs a self-test. This self-test MUST occur after the loading
853 of the previous state. It MAY occur before the entropy collector performs the first
854 update.

4.2.6 SHA-1 Engine

Start of informative comment

The SHA-1, Figure 4:a C5, hash capability is primarily used by the TPM, as it is a trusted implementation of a hash algorithm. The hash interfaces are exposed outside the TPM to support Measurement taking during platform boot phases and to allow environments that have limited capabilities access to a hash functions. The TPM is not a cryptographic accelerator. TCG does not specify minimum throughput requirements for TPM hash services.

End of informative comment

1. The TPM MUST implement the SHA-1 hash algorithm as defined by FIPS-180-1.
2. The output of SHA-1 is 160 bits and all areas that expect a hash value are REQUIRED to support the full 160 bits.
3. The only commands that SHALL be presented to the TPM in-between a TPM_SHA1Start command and a TPM_SHA1Complete command SHALL be a variable number (possibly 0) of TPM_SHA1Update commands.
   a. The TPM_SHA1Update commands can occur in a transport session.
4. Throughout all parts of the specification the characters x1 || x2 imply the concatenation of x1 and x2

4.2.7 Power Detection

Start of informative comment

The power detection component, Figure 4:a C6, manages the TPM power states in conjunction with platform power states. TCG requires that the TPM be notified of all power state changes.

Power detection also supports physical presence assertions. The TPM may restrict command-execution during periods when the operation of the platform is physically constrained. In a PC, operational constraints occur during the power-on self-test (POST) and require Operator input via the keyboard. The TPM might allow access to certain commands while in a constrained execution mode or boot state. At some critical point in the POST process, the TPM may be notified of state changes that affect TPM command processing modes.

End of informative comment

4.2.8 Opt-In

Start of informative comment

The Opt-In component, Figure 4:a C7, provides mechanisms and protections to allow the TPM to be turned on/off, enabled/disabled, activated/deactivated. The Opt-In component maintains the state of persistent and volatile flags and enforces the semantics associated with these flags.

The setting of flags requires either authorization by the TPM Owner or the assertion of physical presence at the platform. The platform's manufacturer determines the techniques used to represent physical-presence. The guiding principle is that no remote entity should

 Copyright © TCG

895 be able to change TPM status without either knowledge of the TPM Owner or the Operator is
896 physically present at the platform. Physical presence may be asserted during a period when
897 platform operation is constrained such as power-up.

898 Non-Volatile Flags:

899 PhysicalPresenceLifetimeLock

900 PhysicalPresenceHWEnable

901 PhysicalPresenceCMDEnable

902 Volatile Flags:

903 PhysicalPresenceV

904 The following truth table explains the conditions in which the PhysicalPresenceV flag may
905 be altered:

Persistent / Volatile	P	P	P	V	
Control Flags	PhysicalPresenceLifetimeLock	PhysicalPresenceHWEnable	PhysicalPresenceCMDEnable	PhysicalPresenceV	
Volatile Access Semantics to Physical Presence Flag	-	F	F	-	No access to PhysicalPresenceV flag.
	-	F	T	T	
	-	-	T	F	Access to PhysicalPresenceV flag through TCS_PhysicalPresence command enabled.
	-	T	-	-	Access to PhysicalPresenceV flag through hardware signal enabled.
	-	T	T	F	Access to PhysicalPresenceV flag through hardware signal or TCS_PhysicalPresence command enabled.
Persistent Access Semantics to Physical Presence Flag	T	F	F	-	Access to PhysicalPresenceV flag permanently disabled.
	T	F	T	T	
	T	F	T	F	Exclusive access to PhysicalPresenceV flag through TCS_PhysicalPresence command permanently enabled.
	T	T	F	-	Exclusive access to PhysicalPresenceV flag through hardware signal permanently enabled.
	T	T	T	F	Access to PhysicalPresenceV flag through hardware signal or TCS_PhysicalPresence command permanently enabled.

906 Table 4:a - Physical Presence Semantics

907 TCG also recognizes the concept of unambiguous physical presence. Conceptually, the use
908 of dedicated electrical hardware providing a trusted path to the Operator has higher
909 precedence than the physicalPresenceV flag value. Unambiguous physical presence may be
910 used to override physicalPresenceV flag value under conditions specified by platform
911 specific design considerations.

912 Additional details relating to physical presence can be found in sections on Volatile and
913 Non-volatile memory.

914 **End of informative comment**

915 4.2.9 Execution Engine

916 **Start of informative comment**

917 The execution engine, Figure 4:a C8, runs program code to execute the TPM commands
918 received from the I/O port. The execution engine is a vital component in ensuring that
919 operations are properly segregated and shield locations are protected.

920 **End of informative comment**

921 4.2.10 Non-Volatile Memory

922 **Start of informative comment**

923 Non-volatile memory component, Figure 4:a C9, is used to store persistent identity and
924 state associated with the TPM. The NV area has set items (like the EK) and also is available
925 for allocation and use by entities authorized by the TPM Owner.

926 The TPM designer should consider the use model of the TPM and if the use of NV storage is
927 a concern. NV storage does have a limited life and using the NV storage in a high volume
928 use model may prematurely wear out the TPM.

929 **End of informative comment**

930 4.3 Data Integrity Register (DIR)

931 **Start of informative comment**

932 The DIR were a version 1.1 function. They provided a place to store information using the
933 TPM NV storage.

934 In 1.2 the DIR are deprecated and the use of the DIR should move to the general purpose
935 NV storage area.

936 The TPM must still support the functionality of the DIR register in the NV storage area.

937 **End of informative comment**

938 1. A TPM MUST provide one Data Integrity Register (DIR)

939 a. The TPM DIR commands are deprecated in 1.2

940 b. The TPM MUST reserve the space for one DIR in the NV storage area

941 c. The TPM MAY have more than 1 DIR.

942 2. The DIR MUST be 160-bit values and MUST be held in TPM shielded-locations.

943 3. The DIR MUST be non-volatile (values are maintained during the power-off state).

944 a. A TPM implementation need not provide the same number of DIRs as PCRs.

945 4.4 Platform Configuration Register (PCR)

946 **Start of informative comment**

 Copyright © TCG

A Platform Configuration Register (PCR) is a 160-bit storage location for discrete integrity measurements. There are a minimum of 16 PCR registers. All PCR registers are shielded-locations and are inside of the TPM. The decision of whether a PCR contains a standard measurement or if the PCR is available for general use is deferred to the platform specific specification.

A large number of integrity metrics may be measured in a platform, and a particular integrity metric may change with time and a new value may need to be stored. It is difficult to authenticate the source of measurement of integrity metrics, and as a result a new value of an integrity metric cannot be permitted to simply overwrite an existing value. (A rogue could erase an existing value that indicates subversion and replace it with a benign value.) Thus, if values of integrity metrics are individually stored, and updates of integrity metrics must be individually stored, it is difficult to place an upper bound on the size of memory that is required to store integrity metrics.

The PCR is designed to hold an unlimited number of measurements in the register. It does this by using a cryptographic hash and hashing all updates to a PCR. The pseudo code for this is:

PCRi New = HASH (PCRi Old value || value to add)

There are two salient properties of cryptographic hash that relate to PCR construction. Ordering – meaning updates to PCRs are not commutative. For example, measuring (A then B) is not the same as measuring (B then A).

The other hash property is one-way-ness. This property means it should be computationally infeasible for an attacker to determine the input message given a PCR value. Furthermore, subsequent updates to a PCR cannot be determined without knowledge of the previous PCR values or all previous input messages provided to a PCR register since the last reset.

End of informative comment

1. The PCR MUST be a 160-bit field that holds a cumulatively updated hash value
2. The PCR MUST have a status field associated with it
3. The PCR MUST be in the RTS and should be in volatile storage
4. The PCR MUST allow for an unlimited number of measurements to be stored in the PCR
5. The PCR MUST preserve the ordering of measurements presented to it
6. A PCR MUST be set to the default value as specified by the PCRReset attribute
7. A TPM implementation MUST provide 16 or more independent PCRs. These PCRs are identified by index and MUST be numbered from 0 (that is, PCR0 through PCR15 are required for TCG compliance). Vendors MAY implement more registers for general-purpose use. Extra registers MUST be numbered contiguously from 16 up to max – 1, where max is the maximum offered by the TPM.
8. The TCG-protected capabilities that expose and modify the PCRs use a 32-bit index, indicating the maximum usable PCR index. However, TCG reserves register indices 230 and higher for later versions of the specification. A TPM implementation MUST NOT provide registers with indices greater than or equal to 230. In this specification, the following terminology is used (although this internal format is not mandated).

988 9. The PSS MUST define at least define one measurement that the RTM MUST make and
989 the PCR where the measurement is stored.

990 10. A TCG measurement agent MAY discard a duplicate event instead of incorporating it in a
991 PCR, provided that:

992 11. A relevant TCG platform specification explicitly permits duplicates of this type of event to
993 be discarded

994 12. The PCR already incorporates at least one event of this type

995 13. An event of this type previously incorporated into the PCR included a statement that
996 duplicate such events may be discarded. This option could be used where frequent
997 recording of sleep states will adversely affect the lifetime of a TPM, for example.

998 14. PCRs and the protected capabilities that operate upon them MAY NOT be used until
999 power-on self-test (TPM POST) has completed. If TPM POST fails, the TPM_Extend
1000 operation will fail; and, of greater importance, the TPM_Quote operation and TPM_Seal
1001 operations that respectively report and examine the PCR contents MUST fail. At the
1002 successful completion of TPM POST, all PCRs MUST be set to their default value (either
1003 0x00…00 or 0xFF…FF). Additionally, the UINT32 flags MUST be set to zero.

Copyright © TCG

5. Endorsement Key Creation

Start of informative comment

The TPM contains a 2048-bit RSA key pair called the endorsement key (EK). The public portion of the key is the PUBEK and the private portion the PRIVEK. Due to the nature of this key pair, both the PUBEK and the PRIVEK have privacy and security concerns.

The TPM has the EK generated before the end customer receives the platform. The Trusted Platform Module Entity (TPME) that causes EK generation is also the entity that will create and sign the EK credential attesting to the validity of the TPM and the EK. The TPME is typically the TPM manufacturer.

The TPM can generate the EK internally using the TPM_CreateEndorsementKey or by using an outside key generator. The EK needs to indicate the genealogy of the EK generation.

Subsequent attempts to either generate an EK or insert an EK must fail.

If the data structure TPM_ENDORSEMENT_CREDENTIAL is stored on a platform after an Owner has taken ownership of that platform, it SHALL exist only in storage to which access is controlled and is available to authorized entities.

End of informative comment

1. The EK MUST be a 2048-bit RSA key
    a. The public portion of the key is the PUBEK
    b. The private portion of the key is the PRIVEK
    c. The PRIVEK SHALL exist only in a TPM-shielded location.
2. Access to the PRIVEK and PUBEK MUST only be via TPM protected capabilities
    a. The protected capabilities MUST require TPM Owner authentication or operator physical presence
3. The generation of the EK may use a process external to the TPM and TPM_CreateEndorsementKeyPair
    a. The external generation MUST result in an EK that has the same properties as an internally generated EK
    b. The external generation process MUST protect the EK from exposure during the generation and insertion of the EK
    c. After insertion of the EK the TPM state MUST be the same as the result of the TPM_CreateEndorsementKeyPair execution
    d. The process MUST guarantee correct generation, cryptographic strength, uniqueness, privacy, and installation into a genuine TPM, of the EK
    e. The entity that signs the EK credential MUST be satisfied that the generation process properly generated the EK and inserted it into the TPM
    f. The process MUST be defined in the target of evaluation (TOE) of the security target in use to evaluate the TPM

1041 ## 5.1 Controlling Access to PRIVEK

1042 **Start of informative comment**

1043 Exposure of the PRIVEK is a security concern.

1044 The TPM must ensure that the PRIVEK is not exposed outside of the TPM

1045 **End of informative comment**

1046 1. The PRIVEK MUST never be out of the control of a TPM shielded location

1047 ## 5.2 Controlling Access to PUBEK

1048 **Start of informative comment**

1049 There are no security concerns with exposure or use of the PUBEK.

1050 Privacy guidelines suggest that PUBEK could be considered personally identifiable
1051 information (PII) if it were associated in some way with personal information (PI) or
1052 associated with other PII, but PUBEK alone cannot be considered PII. Arbitrary random
1053 numbers do not represent a threat to privacy unless further associated with PI or PII. The
1054 PUBEK is an arbitrary random number that may be associated with aggregate platform
1055 information, but not personally identifiable information.

1056 An EK may become associated with personally identifiable information when an alias
1057 platform identifier (AIK) is also associated with PI. The attestation service could include
1058 personal information in the AIK credential, thereby making the AIK-PUBEK association PII –
1059 but not before.

1060 The association of PUBEK with AIK therefore is important to protect via privacy guidelines.
1061 The owner/user of the TPM should be able to control whether PUBEK is disclosed along
1062 with AIK. The owner/user should be notified of personal information that might be added to
1063 an AIK credential, which could result in AIK being considered PII. The owner/user should
1064 be able to evaluate the mechanisms used by an attestation entity to protect PUBEK-AIK
1065 associations before disclosure occurs. No other entity should be privy to owner/user
1066 authorized disclosure besides the intended attestation entity.

1067 Several commands may be used to negotiate the conditions of PUBEK-AIK disclosure.
1068 TPM_MakeIdentity discloses PUBEK-AIK in the context of requesting an AIK credential.
1069 TPM_ActivateIdentity ensures the owner/user has not been spoofed by an interloper. These
1070 interfaces allow the owner/user to choose whether disclosure is acceptable and control the
1071 circumstances under which disclosure takes place. They do not allow the owner/user the
1072 ability to retain control of PUBEK-AIK subsequent to disclosure except by traditional means
1073 of trusting the attestation entity to abide by an acceptable privacy policy. The owner/user is
1074 able to associate the accepted privacy policy with the disclosure operation (e.g.
1075 TPM_MakeIdentity).

1076 A persistent flag called readPubek can be set to TRUE to permit reading of PUBEK via
1077 TPM_ReadPubek. Reporting the PUBEK value is not considered privacy sensitive because it
1078 cannot be associated with any of the AIK keys managed by the TPM without using TPM
1079 protected-capabilities. Keys are encrypted with a nonce when flushed from TPM shielded-
1080 locations, Cryptanalysis of flushed keys will not reveal an association of EK to any AIK.

1081 The command that manipulates the readPubek flag is TPM_DisablePubekRead.

1082 **End of informative comment**

Copyright © TCG

1083 # 6. Attestation Identity Keys

1084 **Start of informative comment**

1085 See 11.4 Attestation Identity Keys.

1086 **End of informative comment**

1087 # 7. TPM Ownership

1088 **Start of informative comment**

1089 Taking ownership of a TPM is the process of inserting a shared secret into a TPM shielded-
1090 location. Any entity that knows the shared secret is a TPM Owner. Proof of ownership
1091 occurs when an entity, in response to a challenge, proves knowledge of the shared secret.
1092 Certain operations in the TPM require authentication from a TPM Owner.

1093 Certain operations also allow the human, with physical possession of the platform, to assert
1094 TPM Ownership rights. When asserting TPM Ownership, using physical presence, the
1095 operations must not expose any secrets protected by the TPM.

1096 The platform owner controls insertion of the shared secret into the TPM. The platform
1097 owner sets the NV persistent flag ownershipEnabled that allows the execution of the
1098 TPM_TakeOwnership command. The TPM_SetOwnerInstall, the command that controls the
1099 value ownershipEnabled, requires the assertion of physical presence.

1100 Attempting to execute TPM_TakeOwnership fails when a TPM already has an owner. To
1101 remove an owner when the current TPM Owner is unable to remove themselves, the human
1102 that is in possession of the platform asserts physical presence and executes
1103 TPM_ForceClear which removes the shared secret.

1104 The insertion protocol that supplies the shared secret has the following requirements:
1105 confidentiality, integrity, remoteness and verifiability.

1106 To provide confidentiality the proposed TPM Owner encrypts the shared secret using the
1107 PUBEK. This requires the PRIVEK to decrypt the value. As the PRIVEK is only available in
1108 the TPM the encrypted shared secret is only available to the intended TPM.

1109 The integrity of the process occurs by the TPM providing proof of the value of the shared
1110 secret inserted into the TPM.

1111 By using the confidentiality and integrity, the protocol is useable by TPM Owners that are
1112 remote to the platform.

1113 The new TPM Owner validates the insertion of the shared secret by using integrity response.

1114 **End of informative comment**

1115 The TPM MUST ship with no Owner installed. The TPM MUST use the ownership-control
1116 protocol (OIAP or OSAP)

1117 ## 7.1 Platform Ownership and Root of Trust for Storage

1118 **Start of informative comment**

1119 The semantics of platform ownership are tied to the Root-of-trust-for-storage (RTS). The
1120 TPM_TakeOwnership command creates a new Storage Root Key (SRK) and new tpmProof
1121 value whenever a new owner is established. It follows that objects owned by a previous
1122 owner will not be inherited by the new owner. Objects that should be inherited must be
1123 transferred by deliberate data migration actions.

1124 **End of informative comment**

8. Authentication and Authorization Data

Start of informative comment

Using security vernacular the terms below apply to the TPM for this discussion:

Authentication: The process of providing proof of claimed ownership of an object or a subject's claimed identity.

Authorization: Granting a subject appropriate access to an object.

Each TPM object that does not allow "public" access contains a 160-bit shared secret. This shared secret is enveloped within the object itself. The TPM grants use of TPM objects based on the presentation of the matching 160-bits using protocols designed to provide protection of the shared secret. This shared secret is called the AuthData.

Neither the TPM, nor its objects (such as keys), contain access controls for its objects (the exception to this is what is provided by the delegation mechanism). If an subject presents the AuthData, that subject is granted full use of the object based on the object's capabilities, not a set of rights or permissions of the subject. This apparent overloading of the concepts of authentication and authorization has caused some confusion. This is caused by having two similarly rooted but distinct perspectives.

From the perspective of the TPM looking out, this AuthData is its sole mechanism for authenticating the owner of its objects, thus from its perspective it is authentication data. However, from the application's perspective this data is typically the result of other functions that might perform authentications or authorizations of subjects using higher level mechanisms such as OS login, file system access, etc. Here, AuthData is a result of these functions so in this usage, it authorizes access to the TPM's objects. From this perspective, i.e., the application looking in on the TPM and its objects, the AuthData is authorization data. For this reason, and thanks to a common root within the English language, the term for this data is chosen to be AuthData and is to be interpreted or expanded as either authentication data or authorization data depending on context and perspective.

The term AuthData refers to the 160-bit value used to either prove ownership of, or authorization to use, an object. This is also called the object's shared secret. The term authorization will be used when referring the combined action of verifying the AuthData and allowing access to the object or function. The term authorization session applies to a state where the AuthData has been authentication and a session handle established that is associated with that authentication.

A wide-range of objects use AuthData. It is used to establish platform ownership, key use restrictions, object migration and to apply access control to opaque objects protected by the TPM.

AuthData is a 160-bit shared-secret plus high-entropy random number. The assumption is the shared-secret and random number are mixed using SHA-1 digesting, but no specific function for generating AuthData is specified by TCG.

TCG command processing sessions (e.g. OSAP, ADIP) may use AuthData as an initialization vector when creating a one-time pad. Session encryption is used to encrypt portions of command messages exchanged between TPM and a caller.

1167 The TPM stores AuthData with TPM controlled-objects and in shielded-locations. AuthData
1168 is never in the clear, when managed by the TPM except in shielded-locations. Only TPM
1169 protected-capabilities may access AuthData (contained in the TPM). AuthData objects may
1170 not be used for any other purpose besides authentication and authorization of TPM
1171 operations on controlled-objects.

1172 Outside the TPM, a reference monitor of some kind is responsible for protecting AuthData.
1173 AuthData should be regarded as a controlled data item (CDI) in the context of the security
1174 model governing the reference monitor. TCG expects this entity to preserve the interests of
1175 the platform Owner.

1176 There is no requirement that instances of AuthData be unique.

1177 **End of informative comment**

1178 The TPM MUST reserve 160 bits for the AuthData. The TPM treats the AuthData as a blob.
1179 The TPM MUST keep AuthData in a shielded-location.

1180 The TPM MUST enforce that the only usage in the TPM of the AuthData is to perform
1181 authorizations.

1182 ## 8.1 Dictionary Attack Considerations

1183 **Start of informative comment**

1184 The decision to provide protections against dictionary attacks is due to the inability of the
1185 TPM to guarantee that an authorization value has high entropy. While the creation and
1186 authorization protocols could change to support the assurance of high entropy values, the
1187 changes would be drastic and would totally invalidate any 1.x TPM version.

1188 Version 1.1 explicitly avoided any requirements for dictionary attack mitigation.

1189 Version 1.2 adds the requirement that the TPM vendor provide some assistance against
1190 dictionary attacks. The internal mechanism is vendor specific. The TPM designer should
1191 review the requirements for dictionary attack mitigation in the Common Criteria.

1192 The 1.2 specification does not provide any functions to turn on the dictionary attack
1193 prevention. The specification does provide a way to reset from the TPM response to an
1194 attack.

1195 By way of example, the following is a way to implement the dictionary attack mitigation.

1196 The TPM keeps a count of failed authorization attempts. The vendor allows the TPM Owner
1197 to set a threshold of failed authorizations. When the count exceeds the threshold, the TPM
1198 locks up and does not respond to any requests for a time out period. The time out period
1199 doubles each time the count exceeds the threshold. If the TPM resets during a time out
1200 period, the time out period starts over after TPM_Init, or TPM_Startup. To reset the count
1201 and the time out period the TPM Owner executes TPM_ResetLockValue. If the authorization
1202 for TPM_ResetLockValue fails, the TPM must lock up for the entire time out period and no
1203 additional attempts at unlocking will be successful. Executing TPM_ResetLockValue when
1204 outside of a time out period still results in the resetting of the count and time out period.

1205 **End of informative comment**

1206 The TPM SHALL incorporate mechanism(s) that will provide some protection against
1207 exhaustive or dictionary attacks on the authorization values stored within the TPM.

1208 This version of the TPM specification does NOT specify the particular strategy to be used.
1209 Some examples might include locking out the TPM after a certain number of failures,
1210 forcing a reboot under some combination of failures, or requiring specific actions on the
1211 part of some actors after an attack has been detected. The mechanisms to manage these
1212 strategies are vendor specific at this time.

1213 If the TPM in response to the attacks locks up for some time period or requires a special
1214 operation to restart, the TPM MUST prevent any authorized TPM command and MAY
1215 prevent any TPM command from executing until the mitigation mechanism completes. The
1216 TPM Owner can reset the mechanism using the TPM_ResetLockValue command.
1217 TPM_ResetLockValue MUST be allowed to run exactly once while the TPM is locked up.

9. TPM Operation

Start of informative comment

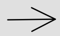
Through the course of TPM operation, it may enter several operational modes that include power-up, self-test, administrative modes and full operation. This section describes TPM operational states and state transition criteria. Where applicable, the TPM commands used to facilitate state transition or function are included in diagrams and descriptions.

The TPM keeps the information relative to the TPM operational state in a combination of persistent and volatile flags. For ease of reading the persistent flags are prefixed by pFlags and the volatile flags prefixed by vFlags.

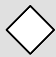
The following state diagram describes TPM operational states at a high level. Subsequent state diagrams drill-down to finer detail that describes fundamental operations, protections on operations and the transitions between them.

The state diagrams use the following notation:

- Signifies a state.

 - Transitions between states are represented as a single headed arrows.

- Circular transitions indicate operations that don't result in a transition to another state.

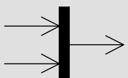
 - Decision boxes split state flow based on a logical test. Decision conditions are called Guards and are identified by bracketed text.

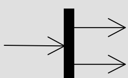
< [text] > Bracketed text indicates transitions that are gated. Text within the brackets describes the pre-condition that must be met before state transition may occur.

< /name > Transitions may list the events that trigger state transition. The forward slash demarcates event names.

- The starting point for reading state diagrams.

- The ending point for state diagrams. Perpetual state systems may not have an ending indicator.

 - The collection bar consolidates multiple identical transition events into a single transition arrow.

 - The distribution bar splits transitions to flow into multiple states.

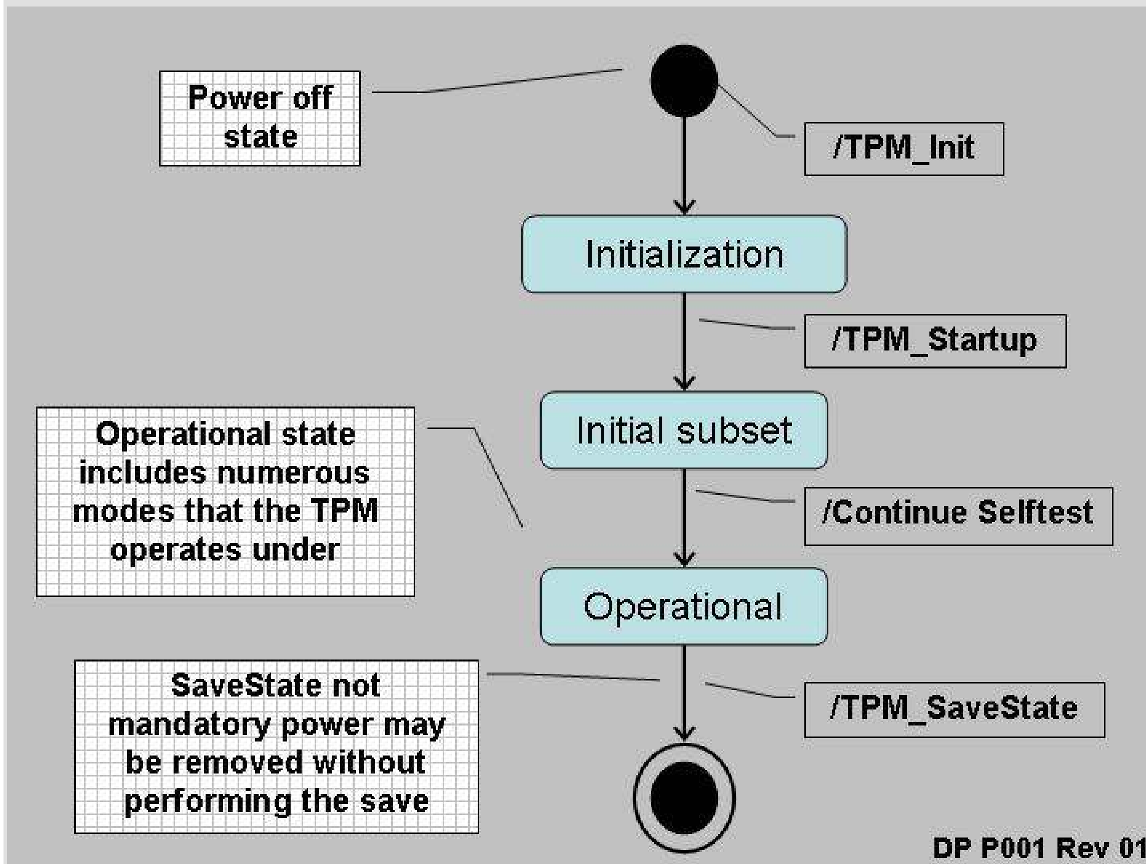
- The history indicator means state values are remembered across context switches or power-cycles.

 Copyright © TCG

1249 **End of informative comment**

1250 9.1 TPM Initialization & Operation State Flow

1251 **Start of informative comment**



1252

1253 **Figure 9:a - TPM Operational States**

1254 **End of informative comment**

1255 9.1.1 Initialization

1256 **Start of informative comment**

1257 TPM_Init transitions the TPM from a power-off state to one where the TPM begins an
1258 initialization process. TPM_Init could be the result of power being applied to the platform or
1259 a hard reset.

1260 TPM_Init sets an internal flag to indicate that the TPM is undergoing initialization. The TPM
1261 must complete initialization before it is operational. The completion of initialization requires
1262 the receipt of the TPM_Startup command.

1263 The TPM is not fully operational until all of the self-tests are complete. Successful
1264 completion of the self-tests allows the TPM to enter fully operational mode.

1265 Fully operational does not imply that all functions of the TPM are available. The TPM needs
1266 to have a TPM Owner and be enabled for all functions to be available.

1267 The TPM transitions out of the operational mode by having power removed from the system.
1268 Prior to the exiting operational mode, the TPM prepares for the transition by executing the
1269 TPM_SaveState command. There is no requirement that TPM_SaveState execute before the
1270 transition to power-off mode occurs.

1271 **End of informative comment**

1272 1. After TPM_Init and until receipt of TPM_Startup the TPM MUST return
1273 TPM_INVALID_POSTINIT for all commands. Prior to receipt of TPM_Startup the TPM
1274 MAY enter shutdown or failure mode.

Copyright © TCG

1275 ## 9.2 Self-Test Modes

1276 **Start of informative comment**

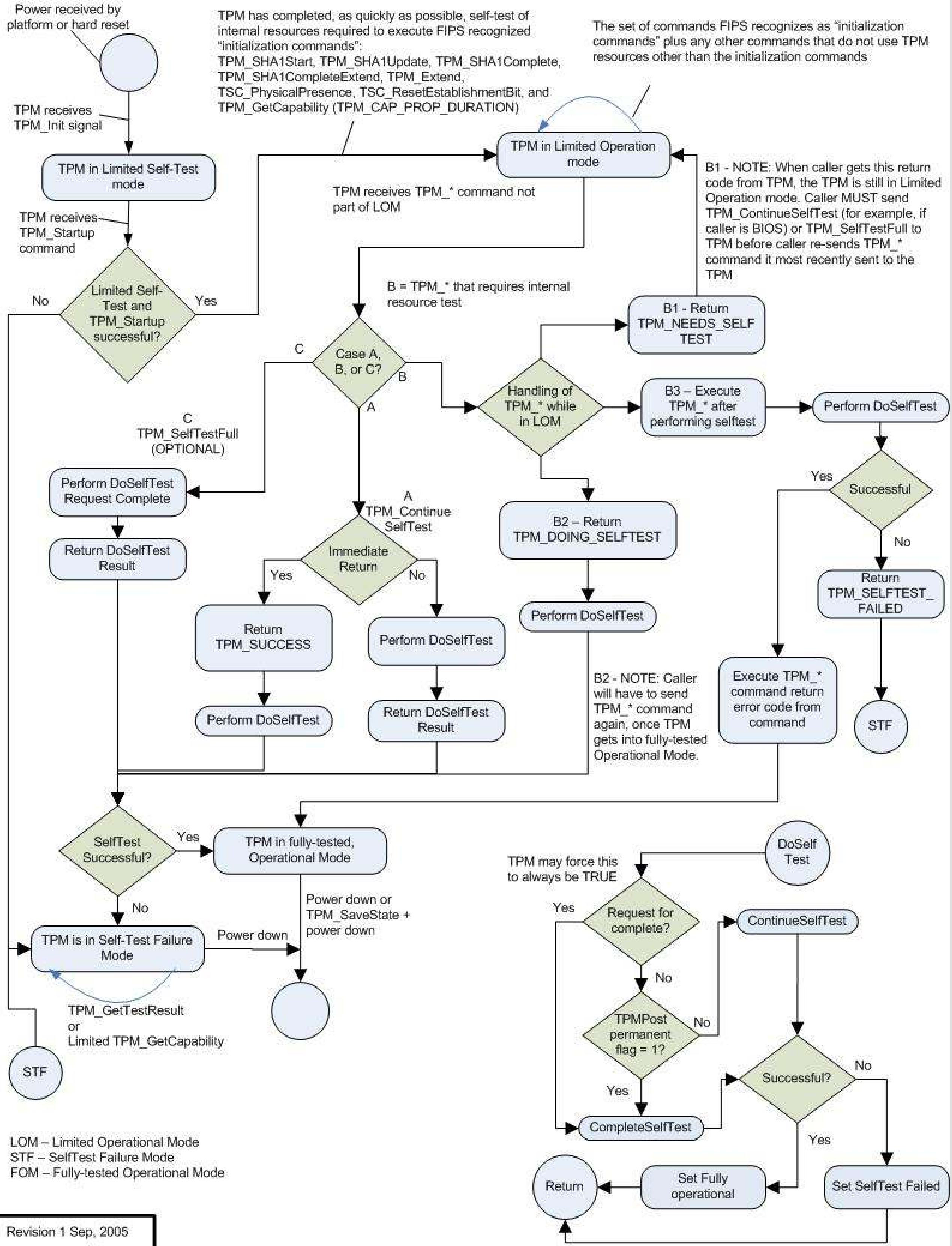


Figure 9:b - Self-Test States

After initialization the TPM performs a limited self-test. This test provides the assurance that a selected subset of TPM commands will perform properly. The limited nature of the self-test allows the TPM to be functional in as short of time as possible. The commands enabled by this self-test are:

TPM_SHA1xxx – Enabling the SHA-1 commands allows the TPM to assist the platform startup code. The startup code may execute in an extremely constrained memory environment and having the TPM resources available to perform hash functions can allow the measurement of code at an early time. While the hash is available, there are no speed requirements on the I/O bus to the TPM or on the TPM itself so use of this functionality may not meet platform startup requirements.

TPM_Extend – Enabling the extend, and by reference the PCR, allows the startup code to perform measurements. Extending could use the SHA-1 TPM commands or perform the hash using the main processor.

TPM_Startup – This command must be available as it is the transition command from the initial environment to the limited operational state.

TPM_ContinueSelfTest – This command causes the TPM to complete the self-tests on all other TPM functions. If TPM receives a command, and the self-test for that command has not been completed, the TPM may implicitly perform the actions of the TPM_ContinueSelfTest command.

TPM_SelfTestFull – A TPM MAY allow this command after initialization, but typically TPM_ContinueSelfTest would be used to avoid repeating the limited self tests.

TPM_GetCapability – A subset of capabilities can be read in the limited operation state.

The complete self-test ensures that all TPM functionality is available and functioning properly.

End of informative comment

1. At startup, a TPM MUST self-test all internal functions that are necessary to do TPM_SHA1Start, TPM_SHA1Update, TPM_SHA1Complete, TPM_SHA1CompleteExtend, TPM_Extend, TPM_Startup, TPM_ContinueSelfTest, a subset of TPM_GetCapability, and TPM_GetTestResult.
2. The TSC_PhysicalPresence and TSC_ResetEstablishmentBit commands do not operate on shielded-locations and have no requirement to be self-tested before any use.
3. The TPM MAY allow TPM_SelfTestFull to be used before completion of the actions of TPM_ContinueSelfTest.
4. The TPM MAY implicitly run the actions of TPM_ContinueSelfTest upon receipt of a command that requires untested resources.
5. The platform specific specification MUST define the maximum startup self-test time.

9.2.1 Operational Self-Test

Start of informative comment

The completion of self-test is initiated by TPM_ContinueSelfTest. The TPM MAY allow TPM_SelfTestFull to be issued instead of TPM_ContinueSelfTest.

1319 TPM_ContinueSelfTest is the command issued during platform initialization after the
1320 platform has made use of the early commands (perhaps for an early measurement), the
1321 platform is now performing other initializations, and the TPM can be left alone to complete
1322 the self-tests. Before any command other than the limited subset is executed, all self-tests
1323 must be complete.

1324 TPM_SelfTestFull is a request to have the TPM perform another complete self-test. This test
1325 will take some time but provides an accurate assessment of the TPM's ability to perform all
1326 operations.

1327 The original design of TPM_ContinueSelfTest was for the TPM to test those functions that
1328 the original startup did not test. The FIPS-140 evaluation of the specification requested a
1329 change such that TPM_ContinueSelfTest would perform a complete self-test. The rationale
1330 is that the original tests are only part of the initialization of the TPM; if they fail, the TPM
1331 does not complete initialization. Performing a complete test after initialization meets the
1332 FIPS-140 requirements. The TPM may work differently in FIPS mode or the TPM may simply
1333 write the TPM_ContinueSelfTest command such that it always performs the complete check.

1334 TPM_ContinueSelfTest causes a test of the TPM internal functions. When
1335 TPM_ContinueSelfTest is asynchronous, the TPM immediately returns a successful result
1336 code before starting the tests. When testing is complete, the TPM does not return any
1337 result. When TPM_ContinueSelfTest is synchronous, the TPM completes the self-tests and
1338 then returns a success or failure result code.

1339 The TPM may reject any command other than the limited subset if self test has not been
1340 completed. Alternatively, the actions of TPM_ContinueSelfTest may start automatically if the
1341 TPM receives a command and there has been no testing of the underlying functionality. If
1342 the TPM implements this implicit self-test, it may immediately return a result code
1343 indicating that it is doing self-test. Alternatively, it may do the self-test, then do the
1344 command, and return only the result code of the command.

1345 Programmers of TPM drivers should take into account the time estimates for self-test and
1346 minimize the polling for self-test completion. While self-test is executing, the TPM may
1347 return an out-of-band "busy" signal to prevent command from being issued. Alternatively,
1348 the TPM may accept the command but delay execution until after the self-test completes.
1349 Either of those alternatives may appear as if the TPM is blocking to upper software layers.
1350 Alternatively, the TPM may return an indication that is doing a self-test.

1351 Upon the completion of the self-tests, the result of the self-tests are held in the TPM such
1352 that a subsequent call to TPM_GetTestResult returns the self-test result.

1353 In version 1.1, there was a separate command to create a signed self-test,
1354 TPM_CertifySelfTest. Version 1.2 deprecates the command. The new use model is to perform
1355 TPM_GetTestResult inside of a transport session and then use
1356 TPM_ReleaseTransportSigned to obtain the signature.

1357 If self-tests fail, the TPM goes into failure state and does not allow most other operations to
1358 continue. The TPM_GetTestResult will operate in failure mode so an outside observer can
1359 obtain information as to the reason for the self-test failure.

1360 A TPM may take three courses of action when presented with a command that requires an
1361 untested resource.

1362 1. The TPM may return TPM_NEEDS_SELFTEST, indicating that the execution of the
1363 command requires TPM_ContinueSelfTest.

2. The TPM may implicitly execute the self-test and return a TPM_DOING_SELFTEST return code, causing the external software to retry the command.
3. The TPM may implicitly execute the self-test, execute the ordinal, and return the results of the ordinal.

The following example shows how software can detect either mechanism with a single piece of code

1. SW sends TPM_xxx command
2. SW checks return code from TPM
3. If return code is TPM_DOING_SELFTEST, SW attempts to resend
   a. If the TIS times out waiting for TPM ready, pause for self-test time then resend
   b. if TIS timeout, then error
4. else if return code is TPM_NEEDS_SELFTEST
   a. Send TPM_ContinueSelfTest
5. else
   a. Process the ordinal return code

End of informative comment

1. The TPM MUST provide startup self-tests. The TPM MUST provide mechanisms to allow the self-tests to be run on demand. The response from the self-tests is pass or fail.
2. The TPM MUST complete the startup self-tests in a manner and timeliness that allows the TPM to be of use to the BIOS during the collection of integrity metrics.
3. The TPM MUST complete the required checks before a given feature is in use. If a function self-test is not complete the TPM MUST return TPM_NEEDS_SELFTEST or TPM_DOING_SELFTEST, or do the self-test before using the feature.
4. There are two sections of startup self-tests: required and recommended. The recommended tests are not a requirement due to time constraints. The TPM manufacturer should perform as many tests as possible within the time constraints.
5. The TPM MUST report the tests that it performs.
6. The TPM MUST provide a mechanism to allow self-test to execute on request by any challenger.
7. The TPM MUST provide for testing of some operations during each execution of the operation.
8. The TPM MUST check the following:
   a. RNG functionality
   b. Reading and extending the integrity registers. The self-test for the integrity registers will leave the integrity registers in a known state.
   c. Testing the EK integrity, if it exists

1400 i. This requirement specifies that the TPM will verify that the endorsement key pair
1401 can encrypt and decrypt a known value. This tests the RSA engine. If the EK has
1402 not yet been generated the TPM action is manufacturer specific.

1403 d. The integrity of the protected capabilities of the TPM

1404 i. This means that the TPM must ensure that its “microcode” has not changed, and
1405 not that a test must be run on each function.

1406 e. Any tamper-resistance markers

1407 i. The tests on the tamper-resistance or tamper-evident markers are under
1408 programmable control. There is no requirement to check tamper-evident tape or
1409 the status of epoxy surrounding the case.

1410 9. The TPM SHOULD check the following:

1411 a. The hash functionality

1412 i. This check will hash a known value and compare it to an expected result. There is
1413 no requirement to accept external data to perform the check.

1414 ii. The TPM MAY support a test using external data.

1415 b. Any symmetric algorithms

1416 i. This check will use known data with a random key to encrypt and decrypt the
1417 data

1418 c. Any additional asymmetric algorithms

1419 i. This check will use known data to encrypt and decrypt.

1420 d. The key-wrapping mechanism

1421 i. The TPM should wrap and unwrap a key. The TPM MUST NOT use the
1422 endorsement key pair for this test.

1423 e. Any other internal mechanisms

1424 10. Self-Test Failure

1425 a. When the TPM detects a failure during any self-test, the TPM MUST enter shutdown
1426 mode. This shutdown mode will allow only the following operations to occur:

1427 i. Update. The update function MAY replace invalid microcode, providing that the
1428 parts of the TPM that provide update functionality have passed self-test.

1429 ii. TPM_GetTestResult. This command can assist the TPM manufacturer in
1430 determining the cause of the self-test failure.

1431 iii. TPM_GetCapability may return limited information as specified in the ordinal.

1432 iv. All other operations will return the error code TPM_FAILEDSELFTEST.

1433 b. The TPM MUST leave failure mode only after receipt of TPM_Init followed by
1434 TPM_Startup(ST_CLEAR).

1435 11. Prior to the completion of the actions of TPM_ContinueSelfTest the TPM MAY respond in
1436 two ways

1437 a. The TPM MAY automatically invoke the actions of TPM_ContinueSelfTest.

Copyright © TCG

1438 i. The TPM MAY return TPM_DOING_SELFTEST.

1439 ii. The TPM may complete the self-test, execute the command, and return the
1440 command result.

1441 b. The TPM MAY return the error code TPM_NEEDS_SELFTEST

1442 9.3 Startup

1443 **Start of informative comment**

1444 Startup transitions the TPM from the initialization state to an operational state. The
1445 transition includes information from the platform to inform the TPM of the platform
1446 operating state. TPM_Startup has three options: Clear, State and Deactivated.

1447 The Clear option informs the TPM that the platform is starting in a “cleared” state or most
1448 likely a complete reboot. The TPM is to set itself to the default values and operational state
1449 specified by the TPM Owner.

1450 The State option informs the TPM that the platform is requesting the TPM to recover a saved
1451 state and continue operation from the saved state. The platform previously made the
1452 TPM_SaveState request to the TPM such that the TPM prepares values to be recovered later.

1453 The Deactivated state informs the TPM that it should not allow further operations and
1454 should fail all subsequent command requests. The Deactivated state can only be reset by
1455 performing another TPM_Init.

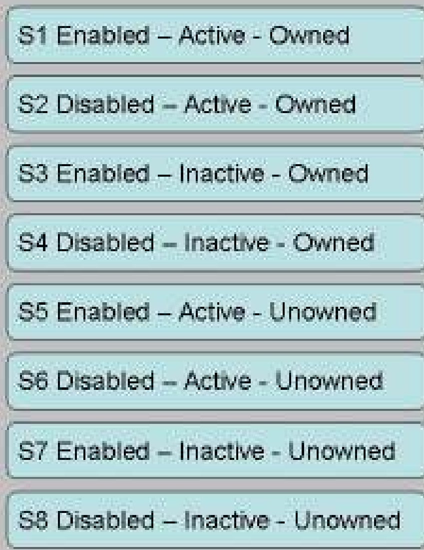
1456 **End of informative comment**

1457 9.4 Operational Mode

1458 **Start of informative comment**

1459 After the TPM completes both TPM_Startup and self-tests, the TPM is ready for operation.

1460 There are three discrete states, enabled or disabled, active or inactive and owned or
1461 unowned. These three states when combined form eight operational modes.



1462

1463 Figure 9:c - Eight Modes of Operation

1464 S1 is the fully operational state where all TPM functions are available. S8 represents a mode
1465 where all TPM features (except those to change the state) are off.

1466 Given the eight modes of operation, the TPM can be flexible in accommodating a wide range
1467 of usage scenarios. The default delivery state for a TPM should be S8 (disabled, inactive and
1468 unowned). In S8, the only mechanism available to move the TPM to S1 is having physical
1469 access to the platform.

1470 Two examples illustrate the possibilities of shipping combinations.

1471 Example 1

1472 The customer does not want the TPM to attest to any information relative to the platform.
1473 The customer does not want any remote entity to attempt to change the control options that
1474 the platform owner is setting. For this customer the platform manufacturer sets the TPM in
1475 S8 (disabled, deactivated and unowned).

1476 To change the state of the platform the platform owner would assert physical presence and
1477 enable, activate and insert the TPM Owner shared secret. The details of how to change the
1478 various modes is in subsequent sections.

1479 This particular sequence gives maximum control to the customer.

1480 Example 2

1481 A corporate customer wishes to have platforms shipped to their employees and the IT
1482 department wishes to take control of the TPM remotely. To satisfy these needs the TPM
1483 should be in S5 (enabled, active and unowned). When the platform connects to the
1484 corporate LAN the IT department would execute the TPM_TakeOwnership command
1485 remotely.

1486 This sequence allows the IT department to accept platforms into their network without
1487 having to have physical access to each new machine.

1488 **End of informative comment**

1489 The TPM MUST have commands to perform the following:

1490 1. Enable and disable the TPM. These commands MUST work as TPM Owner authorized or
1491 with the assertion of physical presence

1492 2. Activate and deactivate the TPM. These commands MUST work as TPM Owner
1493 authorized or with the assertion of physical presence

1494 3. Activate and deactivate the ability to take ownership of the TPM

1495 4. Assert ownership of the TPM.

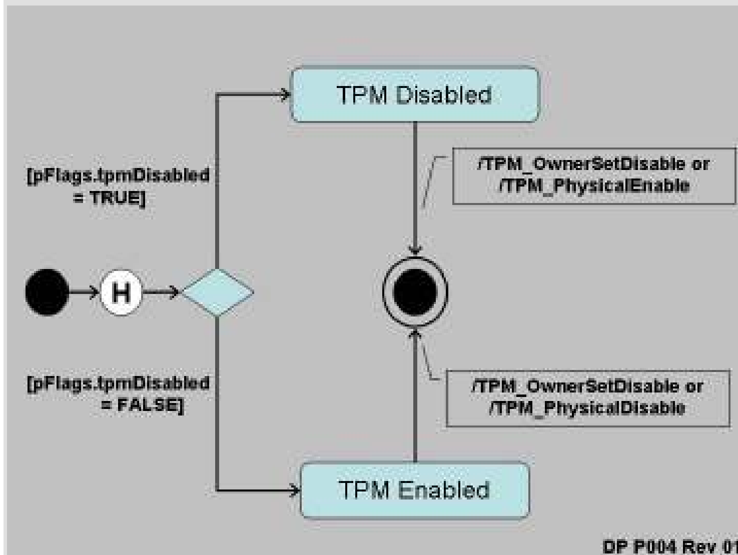
1496 9.4.1 Enabling a TPM

1497 **Informative comment**

1498 A disabled TPM is not able to execute commands that use the resources of a TPM. While
1499 some commands are available (SHA-1 for example) the TPM is not able to load keys and
1500 perform TPM_Seal and other such operations. These restrictions are the same as for an
1501 inactive TPM. The difference between inactive and disabled is that a disabled TPM is unable

Copyright © TCG

1502 to execute the TPM_TakeOwnership command. A disabled TPM that has a TPM Owner is not
1503 able to execute normal TPM commands.

1504 

1505 pFlags.tpmDisabled contains the current enablement status. When set to TRUE the TPM is
1506 disabled, when FALSE the TPM is enabled.

1507 Changing the setting pFlags.tpmDisabled has no effect on any secrets or other values held
1508 by the TPM. No keys, monotonic counters or other resources are invalidated by changing
1509 TPM enablement. There is no guarantee that session resources (like transport sessions)
1510 survive the change in enablement, but there is no loss of secrets.

1511 The TPM_OwnerSetDisable command can be used to transition in either Enabled or
1512 Disabled states. The desired state is a parameter to TPM_OwnerSetDisable. This command
1513 requires TPM Owner authentication to operate. It is suitable for post-boot and remote
1514 invocation.

1515 An unowned TPM requires the execution of TPM_PhysicalEnable to enable the TPM and
1516 TPM_PhysicalDisable to disable the TPM. Operators of an owned TPM can also execute
1517 these two commands. The use of the physical commands allows a platform operator to
1518 disable the TPM without TPM Owner authorization.

1519 TPM_PhysicalEnable transitions the TPM from Disabled to Enabled state. This command is
1520 guarded by a requirement of operator physical presence. Additionally, this command can be
1521 invoked by a physical event at the platform, whether or not the TPM has an Owner or there
1522 is a human physically present. This command is suitable for pre-boot invocation.

1523 TPM_PhysicalDisable transitions the TPM from Enabled to Disabled state. It has the same
1524 guard and invocation properties as TPM_PhysicalEnable.

1525 The subset of commands the TPM is able to execute is defined in the structures document
1526 in the persistent flag section.

1527 Misuse of the disabled state can result in denial-of-service. Proper management of Owner
1528 AuthData and physical access to the platform is a critical element in ensuring availability of
1529 the system.

1530 **End of informative comment**

1. The TPM MUST provide an enable and disable command that is executed with TPM Owner authorization.
2. The TPM MUST provide an enable and disable command this is executed locally using physical presence.

9.4.2 Activating a TPM

Informative comment

A deactivated TPM is not able to execute commands that use TPM resources. A major difference between deactivated and disabled is that a deactivated TPM CAN execute the TPM_TakeOwnership command.

Deactivated may be used to prevent the (obscure) attack where a TPM is readied for TPM_TakeOwnership but a remote rogue manages to take ownership of a platform just before the genuine owner, and immediately has use of the TPM's facilities. To defeat this attack, a genuine owner should set disable==FALSE, ownership==TRUE, deactivate==TRUE, execute TPM_takeOwnership, and then set deactivate==FALSE after verifying that the genuine owner is the actual TPM owner.

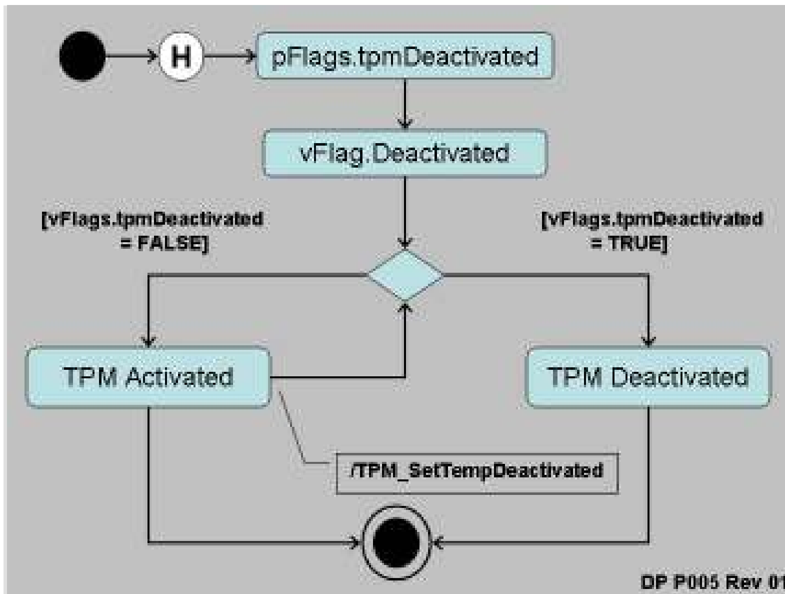
Activation control is with both persistent and volatile flags. The persistent flag is never directly checked by the TPM, rather it is the source of the original setting for the volatile flag. During TPM initialization the value of pFlags.tpmDeactivated is copied to vFlags.tpmDeactivated. When the TPM execution engine checks for TPM activation, it only references vFlags.tpmDeactivated.

Toggling the state of pFlags.tpmDeactivated uses TPM_PhysicalSetDeactivated. This command requires physical presence. There is no associated TPM Owner authenticated command as the TPM Owner can always execute TPM_OwnerSetDisabled which results in the same TPM operations. The toggling of this flag does not affect the current operation of the TPM but requires a reboot of the platform such that the persistent flag is again copied to the volatile flag.

The volatile flag, vFlags.tpmDeactivated, is set during initialization by the value of pFlags.tpmDeactivated. If vFlags.tpmDeactivated is TRUE the only way to reactivate the TPM is to reboot the platform and have pFlags reset the vFlags value.

If vFlags.tpmDeactivated is FALSE, running TPM_SetTempDeactivated will set vFlags.tpmDeactivated to TRUE and then require a reboot of the platform to reactivate the platform.

Copyright © TCG

1563 

1564 Figure 9:d - Activated and Deactivated States

1565 TPM activation is for Operator convenience. It allows the operator to deactivate the platform
1566 (temporarily, using TPM_SetTempDeactivated) during a user session when the operator does
1567 not want to disclose platform or attestation identity. This provides operator privacy, since
1568 PCRs could provide cryptographic proof of an operation. PCRs are inaccessible when a TPM
1569 is deactivated. They cannot be used for authorization, nor can they be read. The reboot
1570 required to activate a TPM also resets the PCRs.

1571 The subset of commands that are available when the TPM is deactivated is contained in the
1572 structures document. The TPM_TakeOwnership command is available when deactivated.
1573 The TPM_Extend command is available when deactivated so that software (e.g. a BIOS) can
1574 run the command without the need to handle an error. The PCR extend operation is
1575 irrelevant, since the resulting PCR value cannot be used.

1576 **End of informative comment**

1577 1. The TPM MUST maintain a non-volatile flag that indicates the activation state

1578 2. The TPM MUST provide for the setting of the non-volatile flag using a command that
1579 requires physical presence

1580 3. The TPM MUST sets a volatile flag using the current setting of the non-volatile flag.

1581 4. The TPM MUST provide for a command that deactivates the TPM immediately

1582 5. The only mechanism to reactivate a TPM once deactivated is to power-cycle the system.

1583 9.4.3 Taking TPM Ownership

1584 **Start of informative comment**

1585 The owner of the TPM has ultimate control of the TPM. The owner of the TPM can enable or
1586 disable the TPM, create AIK and set policies for the TPM. The process of taking ownership
1587 must be a tightly controlled process with numerous checks and balances.

The protections around the taking of ownership include the enablement status, specific persistent flags and the assertion of physical presence.

Control of the TPM revolves around knowledge of the TPM Owner authentication value. Proving knowledge of authentication value proves the calling entity is the TPM Owner. It is possible for more than one entity to know the TPM Owner authentication value.

The TPM provides no mechanisms to recover a lost TPM Owner authentication value.

Recovery from a lost or forgotten TPM Owner authentication value involves removing the old value and installing a new one. The removal of the old value invalidates all information associated with the previous value. Insertion of a new value can occur after the removal of the old value.

A disabled and inactive TPM that has no TPM Owner cannot install an owner.

To invalidate the TPM Owner authentication value use either TPM_OwnerClear or TPM_ForceClear.

End of informative comment

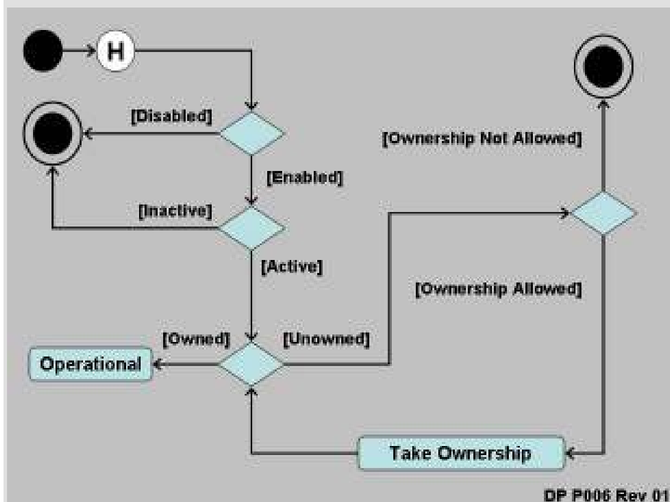
1. The TPM Owner authentication value MUST be a 160-bits
2. The TPM Owner authentication value MUST be held in persistent storage
3. The TPM MUST have no mechanisms to recover a lost TPM Owner authentication value

9.4.3.1 Enabling Ownership

Informative comment

The state that a TPM must be in to allow for TPM_TakeOwnership to succeed is; enabled and fFlags.OwnershipEnabled TRUE.

The following diagram shows the states and the operational checks the TPM makes before allowing the insertion of the TPM Ownership value.



The TPM checks the disabled flag and then the inactive flag. If the flags indicate enabled then the TPM checks for the existence of a TPM Owner. If an Owner is not present the TPM

Copyright © TCG

1615 then checks the OwnershipDisabled flag. If TRUE the TPM_TakeOwnership command will
1616 execute.

1617 While the TPM has no Owner but is enabled and active there is a limited subset of
1618 commands that will successfully execute.

1619 The TPM_SetOwnerInstall command toggles the state of the pFlags.OwnershipDisabled.
1620 TPM_SetOwnerInstall requires the assertion of physical presence to execute.

1621 **End of informative comment**

1622 9.4.4 Transitioning Between Operational States

1623 **Start of informative comment**

1624 The following table is a recap of the commands necessary to transition a TPM from one state
1625 to another.

State	TPM Owner Auth	Physical Presence	Persistence
Disabled to Enabled	TPM_OwnerSetDisable	TPM_PhysicalEnable	permanent
Enabled to Disabled	TPM_OwnerSetDisable	TPM_PhysicalDisable	permanent
Inactive to Active		TPM_PhysicalSetDeactivated	permanent
Active to Inactive		TPM_PhysicalSetDeactivated	permanent
Active to Inactive		TPM_SetTempDeactivated	boot cycle

1626

1627 **End of informative comment**

1628 9.5 Clearing the TPM

1629 **Start of informative comment**

1630 Clearing the TPM is the process of returning the TPM to factory defaults. It is possible the
1631 platform owner will change when in this state.

1632 The commands to clear a TPM require either TPM Owner authentication or the assertion of
1633 physical presence.

1634 The clear process performs the following tasks:

1635 Invalidate the SRK. Once invalidated all information stored using the SRK is now
1636 unavailable. The invalidation does not change the blobs using the SRK rather there is no
1637 way to decrypt the blobs after invalidation of the SRK.

1638 Invalidate tpmProof. tpmProof is a value that provides the uniqueness to values stored off of
1639 the TPM. By invalidating tpmProof all off TPM blobs will no longer load on the TPM.

1640 Invalidate the TPM Owner authentication value. With the authentication value invalidated
1641 there are no TPM Owner authenticated commands that will execute.

1642 Reset volatile and non-volatile data to manufacturer defaults.

1643 The clear must not affect the EK.

1644 Once cleared the TPM will return TPM_NOSRK to commands that require authentication.

1645 The PCR values are undefined after a clear operation. The TPM must go through TPM_Init to
1646 properly set the PCR values.

1647 Clear authentication comes from either the TPM owner or the assertion of physical
1648 presence. As the clear commands present a real opportunity for a denial of service attack
1649 there are mechanisms in place disabling the clear commands.

1650 Disabling TPM_OwnerClear uses the TPM_DisableOwnerClear command. The state of ability
1651 to execute TPM_OwnerClear is then held as one of the non-volatile flags.

1652 Enablement of TPM_ForceClear is held in the volatile disableForceClear flag.
1653 disableForceClear is set to FALSE during TPM_Init. To disable the command software
1654 should issue the TPM_DisableForceClear command.

1655 During the TPM startup processing anyone with physical access to the machine can issue
1656 the TPM_ForceClear command. This command performs the clear operations if it has not
1657 been disabled by vFlags.DisabledForceClear being TRUE.

1658 The TPM can be configured to block all forms of clear operations. It is advisable to block
1659 clear operations to prevent an otherwise trivial denial-of-service attack. The assumption is
1660 the system startup code will issue the TPM_DisableForceClear on each power-cycle after it
1661 is determined the TPM_ForceClear command will not be necessary. The purpose of the
1662 TPM_ForceClear command is to recover from the state where the Owner has lost or
1663 forgotten the TPM Owner-authentication-data.

1664 The TPM_ForceClear must only be possible when the issuer has physical access to the
1665 platform. The manufacturer of a platform determines the exact definition of physical access.

1666 The commands to clear a TPM require either TPM Owner authentication, TPM_OwnerClear,
1667 or the assertion of physical presence, TPM_ForceClear.

1668 **End of informative comment**

1669 1. The TPM MUST support the clear operations.

1670 a. Clear operations MUST be authenticated by either the TPM Owner or physical
1671 presence

1672 b. The TPM MUST support mechanisms to disable the clear operations

1673 2. The clear operation MUST perform at least the following actions

1674 a. SRK invalidation

1675 b. tpmProof invalidation

1676 c. TPM Owner authentication value invalidation

1677 d. Resetting non-volatile values to defaults

1678 e. Invalidation of volatile values

1679 f. Invalidation of internal resources

1680 3. The clear operation must not affect the EK.

Copyright © TCG

10. Physical Presence

Start of informative comment

This specification describes commands that require physical presence at the platform before the command will operate. Physical presence implies direct interaction by a person – i.e. Operator with the platform / TPM.

The type of controls that imply special privilege include:

- Clearing an existing Owner from the TPM,
- Temporarily deactivating a TPM,
- Temporarily disabling a TPM.

Physical presence implies a level of control and authorization to perform basic administrative tasks and to bootstrap management and access control mechanisms.

Protection of low-level administrative interfaces can be provided by physical and electrical methods; or by software; or a combination of both. The guiding principle for designers is the protection mechanism should be difficult or impossible to spoof by rogue software. Designers should take advantage of restricted states inherent in platform operation. For example, in a PC, software executed during the power-on self-test (POST) cannot be disturbed without physical access to the platform. Alternatively, a hardware switch indicating physical presence is very difficult to circumvent by rogue software or remote attackers.

TPM and platform manufacturers will determine the actual implementation approach. The strength of the protection mechanisms is determined by an evaluation of the platform.

Physical presence indication is implemented as a flag in volatile memory known as the PhysicalPresenceV flag. When physical presence is established (TRUE) several TPM commands are able to function. They include:

TPM_PhysicalEnable,

TPM_PhysicalDisable,

TPM_PhysicalSetDeactivated,

TPM_ForceClear,

TPM_SetOwnerInstall,

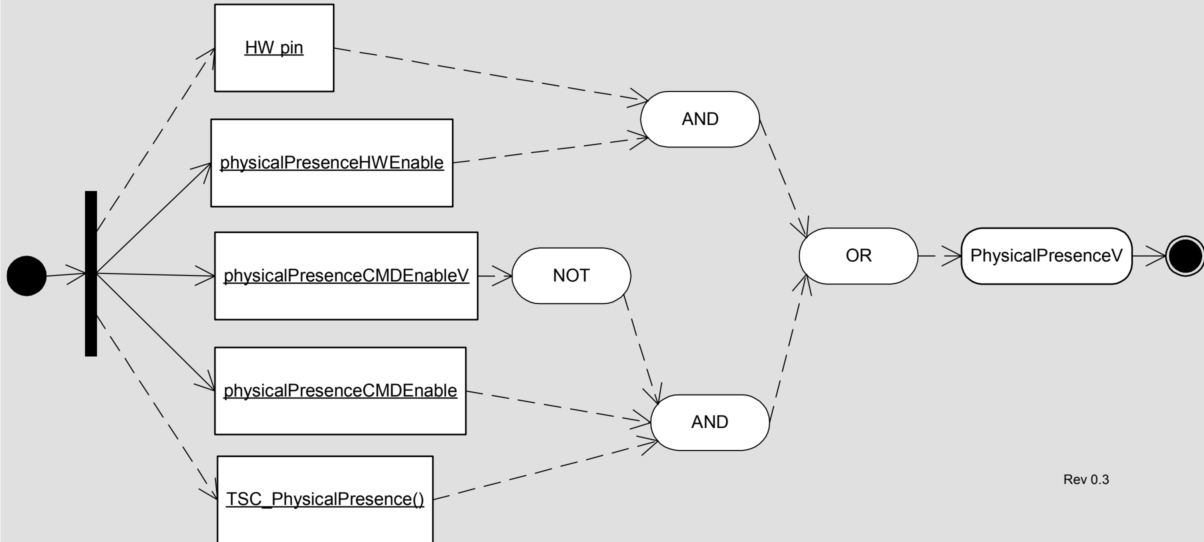
In order to execute these commands, the TPM must obtain unambiguous assurance that the operation is authorized by physical-presence at the platform. The command processor in the I/O component checks the physicalPresenceV flag before continuing processing of TPM command blocks. The volatile physicalPresenceV flag is set only while the Operator is indeed physically present.

TPM designers should take precautions to ensure testing of the physicalPresenceV flag value is not mask-able. For example, a special bus cycle could be used or a dedicated line implemented.

There is an exception to physical presence semantics that allows a remote entity the ability to assert physical presence when that entity is not physically present. The TSC_PhysicalPresence command is used to change polarity of the physicalPresenceV flag.

1721 Its use is heavily guarded. See sections describing the TPM Opt-In component; and Volatile
1722 and Non-volatile memory components.

1723 The following diagram illustrates the flow of logic controlling updates to the
1724 physicalPresenceV flag:



1725

1726 Figure 10:a - Physical Presence Control Logic

1727 This diagram shows that the vFlags.physicalPresenceV flag may be updated either by a HW
1728 pin or through the TSC_PhysicalPresence command, but gated by persistent control flags
1729 and a temporal lock. Observe, the reverse logic surrounding the use of
1730 TSC_PhysicalPresence command. When the physicalPresenceCMDEnable flag is set and the
1731 physicalPresenceCMDEnableV is not set, and the TSC_PhysicalPresence command may
1732 execute.

1733 The physicalPresenceV flag may be overridden by unambiguous physical presence.
1734 Conceptually, the use of dedicated electrical hardware providing a trusted path to the
1735 Operator has higher precedence than the physicalPresenceV flag value. Implementers
1736 should take this into consideration when implementing physical presence indicators.

1737 **End of informative comment**

1738 1. The requirement for physical presence MUST be met by the platform manufacturer
1739 using some physical mechanism.

1740 2. It SHALL be impossible to intercept or subvert indication of physical presence to the
1741 TPM by the execution of software on the platform.

Copyright © TCG

1742 # 11. Root of Trust for Reporting (RTR)

1743 **Start of informative comment**

1744 The RTR is responsible for establishing platform identities, reporting platform
1745 configurations, protecting reported values, and providing a function for attesting to reported
1746 values. The RTR shares responsibility of protecting measurement digests with the RTS.

1747 The interaction between the RTR and RTS is a critical component. The design and
1748 implementation of the interaction between the RTR and RTS should mitigate observation
1749 and tampering with the messages. It is strongly encouraged that the RTR and RTS
1750 implementation occur in the same package such there are no external observation points.
1751 For a silicon based TPM this would imply that the RTR and RTS are in the same silicon
1752 package with no external busses.

1753 **End of informative comment**

1754 1. An instantiation of the RTS and RTR SHALL do the following:

1755 a. Be resistant to all forms of software attack and to the forms of physical attack
1756 implied by the platform's Protection Profile

1757 b. Supply an accurate digest of all sequences of presented integrity metrics

1758 ## 11.1 Platform Identity

1759 **Start of informative comment**

1760 The RTR is a cryptographic identity used to distinguish and authenticate an individual
1761 TPM. The TPM uses the RTR to answer an integrity challenge.

1762 In the TPM, the Endorsement Key (EK) is the RTR. The EK is cryptographically unique and
1763 bound to the TPM.

1764 Prior to any use of the TPM, the RTR must be instantiated. Instantiation may occur during
1765 TPM manufacturing or platform manufacturing. The business issues and manufacturing
1766 flow determines how a specific TPM and platform is initialized with the EK.

1767 As the RTR is cryptographically unique, the use of the RTR must only occur in controlled
1768 circumstances due to privacy concerns. The EK is only available for two operations:
1769 establishing the TPM Owner and establishing Attestation Identity Key (AIK) values and
1770 credentials. There is a prohibition on the use of the EK for any other operation.

1771 **End of informative comment**

1772 1. The RTR MUST have a cryptographic identity.

1773 a. The cryptographic identity of the RTR is the Endorsement Key (EK).

1774 2. The EK MUST be

1775 a. Statistically unique

1776 i. When the TPM is in FIPS mode, the EK MUST be generated using a random
1777 number generator that meets FIPS requirements.

1778 ii. Difficult to forge or counterfeit

1779 b. Verifiable during the AIK creation process

3. The EK SHALL only participate in
    a. TPM Ownership insertion
    b. AIK creation and verification

11.2 RTR to Platform Binding

Start of informative comment

When performing validation of the EK and the platform the challenger wishes to have knowledge of the binding of RTR to platform. The RTR is bound to a TPM hence if the platform can show the binding of TPM to platform the challenger can reasonably believe the RTR and platform binding.

The TPM cannot provide all of the information necessary for the challenger to trust in the binding. That information comes from the manufacturing process and occurs outside the control of the TPM.

End of informative comment

1. The EK is transitively bound to the Platform via the TPM as follows:
    a. An EK is bound to one and only one TPM (i.e., there is a one to one correspondence between an Endorsement Key and a TPM.)
    b. A TPM is bound to one and only one Platform. (i.e., there is a one to one correspondence between a TPM and a Platform.)
    c. Therefore, an EK is bound to a Platform. (i.e., there is a one to one correspondence between an Endorsement Key and a Platform.)

11.3 Platform Identity and Privacy Considerations

Start of informative comment

The uniqueness property of cryptographic identities raises concerns that use of that identity could result in aggregation of activity logs. Analysis of the aggregated activity could reveal personal information that a user of a platform would not otherwise approve for distribution to the aggregators. Both EK and AIK identities have this property.

To counter undesired aggregation, TCG encourages the use of domain specific AIK keys and restricts the use of the EK key. The platform owner controls generation and distribution of AIK public keys.

If a digital signature was performed by the EK, then any entity could track the use of the EK. So use of the EK as a signature is cryptographically sound, but this does not ensure privacy. Therefore, a mechanism to allow verifiers (human or machine) to determine that the TPM really signed the message without using the EK is required.

End of informative comment

11.4 Attestation Identity Keys

Start of informative comment

An Attestation Identity Key (AIK) is an alias for the Endorsement Key (EK). The EK cannot perform signatures for security reasons and due to privacy concerns.

1818 Generation of an AIK can occur anytime after establishment of the TPM Owner. The TPM
1819 can create a virtually unlimited number of AIK.

1820 The TPM Owner controls all aspects of the generation and activation of an AIK. The TPM
1821 Owner controls any data associated with the AIK. The AIK credential may contain
1822 application specific information. The AIK must contain identification such that the TPM can
1823 properly enforce the restrictions placed on an AIK.

1824 The AIK is an asymmetric key pair. For interoperability, the AIK is an RSA 2048-bit key. The
1825 TPM must protect the private portion of the asymmetric key and ensure that the value is
1826 never exposed. The user of an AIK must prove knowledge of the 160-bit AIK authorization
1827 value to use the AIK.

1828 An AIK is a signature key, and is never used for encryption. It only signs information
1829 generated internally by the TPM. The data could include PCR, other keys and TPM status
1830 information. The AIK must never sign arbitrary external data, since it would be possible for
1831 an attacker to create a block of data that appears to be a PCR value.

1832 AIK creation involves two TPM commands.

1833 The TPM_MakeIdentity command causes the TPM to generate the AIK key pair. The
1834 command also discloses the EK-AIK binding to the service that will issue the AIK credential.

1835 The TPM_ActivateIdentity command unwraps a session key that allows for the decryption of
1836 the AIK credential. The session key was encrypted using the PUBEK and requires the
1837 PRIVEK to perform the decryption.

1838 Use of the AIK credential is outside of the control of the TPM.

1839 **End of informative comment**

1840 1. The TPM MUST permanently mark an AIK such that, for all subsequent uses of the AIK,
1841 the AIK restrictions are enforced.

1842 2. An AIK MUST be:

1843 a. Statistically unique

1844 b. Difficult to forge or counterfeit

1845 c. Verifiable to challengers

1846 3. For interoperability the AIK MUST be

1847 a. An RSA 2048-bit key

1848 4. The AIK MUST only sign data generated by the TPM

1849 ### 11.4.1 AIK Creation

1850 **Start of informative comment**

1851 As the AIK is an alias for the EK. The AIK creation process requires TPM Owner
1852 authorization. The process actually requires two TPM Owner authorizations; creation and
1853 credential activation.

1854 The AIK credential creation process is outside the control of the TPM. However, the
1855 certification authority (CA) will attest (with the AIK credential) that the AIK is tied to valid
1856 Endorsement, Platform and Conformance credentials.

1857 Without these credentials, the AIK cannot prove that PCR values belong to a TPM. An owner
1858 may decide to trust any key generated by TPM_MakeIdentity without activating the identity
1859 (e.g., because he is an administrator in a controlled company environment). In this case,
1860 the owner needs no credential. Another challenger can only trust that the AIK belongs to a
1861 TPM by seeing the credential of a trustworthy CA.

1862 **End of informative comment**

1863 1. The TPM Owner MUST authorize the AIK creation process.

1864 2. The TPM MUST use a protected function to perform the AIK creation.

1865 3. The TPM Owner MUST indicate the entity that will provide the AIK credential as part of
1866 the AIK creation process.

1867 4. The TPM Owner MAY indicate that NO credential will ever be created. If the TPM Owner
1868 does indicate that no credential will be provided the TPM MUST ensure that no
1869 credential can be created.

1870 5. The TTP MAY apply policies to determine if the presented AIK should be granted a
1871 credential.

1872 6. The credential request package MUST be useable by only the Privacy CA selected by the
1873 TPM Owner.

1874 7. The AIK credential MUST be only obtainable by the TPM that created the AIK credential
1875 request.

1876
11.4.2 AIK Storage

1877 **Start of informative comment**

1878 The AIK may be stored on some general-purpose storage device.

1879 When held outside of the TPM the AIK sensitive data must be encrypted and integrity
1880 protected.

1881 **End of informative comment**

1882 1. When held outside of the TPM AIK encryption and integrity protection MUST protect the
1883 AIK sensitive information

1884 2. The migration of AIK from one TPM to another MUST be prohibited

 Copyright © TCG

1885 # 12. Root of Trust for Storage (RTS)

1886 **Start of informative comment**

1887 The RTS provides protection on data in use by the TPM but held in external storage devices.
1888 The RTS provides confidentiality and integrity for the external blobs.

1889 The RTS also provides the mechanism to ensure that the release of information only occurs
1890 in a named environment. The naming of an environment uses the PCR selection to
1891 enumerate the values.

1892 Data protected by the RTS can migrate to other TPM.

1893 **End of informative comment**

1894 1. The number and size of values held by the RTS SHOULD be limited only by the volume
1895 of storage available on the platform

1896 2. The TPM MUST ensure that TPM_PERMANENT_DATA -> tpmProof is only inserted into
1897 TPM internally generated and non-migratable information.

1898 ## 12.1 Loading and Unloading Blobs

1899 **Start of informative comment**

1900 The TPM provides several commands to store and load RTS controlled data.

	Class	Command	Analog	Comment
1	Data / Internal / TPM	TPM_MakeIdentity	TPM_ActivateIdentity	Special purpose data
2	Data / External / TPM	TSS_Bind	TPM_Unbind	
3	Data / Internal / PCR	TPM_Seal	TPM_Unseal	
4	Data / External / PCR			
5	Key / Internal / TPM	TPM_CreateWrapKey	TPM_LoadKey	
6	Key / External / TPM	TSS_WrapKey	TPM_LoadKey	
7	Key / Internal / PCR			
8	Key / External / PCR	TSS_WrapKeyToPcr	TPM_LoadKey	

1901 # 13. Transport Sessions and Authorization Protocols

1902 **Start of informative comment**

1903 The purpose of the authorization protocols and mechanisms is to prove to the TPM that the
1904 requestor has permission to perform a function and use some object. The proof comes from
1905 the knowledge of a shared secret.

1906 AuthData is available for the TPM Owner and each entity (keys, for example) that the TPM
1907 controls. The AuthData for the TPM Owner and the SRK are held within the TPM itself and
1908 the AuthData for other entities are held with the entity.

1909 The TPM Owner AuthData allows the Owner to prove ownership of the TPM. Proving
1910 ownership of the TPM does not immediately allow all operations – the TPM Owner is not a
1911 "super user" and additional AuthData must be provided for each entity or operation that
1912 has protection.

1913 The TPM treats knowledge of the AuthData as complete proof of ownership of the entity. No
1914 other checks are necessary. The requestor (any entity that wishes to execute a command on
1915 the TPM or use a specific entity) may have additional protections and requirements where
1916 he or she (or it) saves the AuthData; however, the TPM places no additional requirements.

1917 There are three protocols to securely pass a proof of knowledge of AuthData from requestor
1918 to TPM; the "Object-Independent Authorization Protocol" (OIAP), the "Object-Specific
1919 Authorization Protocol" (OSAP) and the "Delegate-Specific Authorization Protocol" (DSAP).
1920 The OIAP supports multiple authorization sessions for arbitrary entities. The OSAP
1921 supports an authentication session for a single entity and enables the confidential
1922 transmission of new authorization information. The DSAP supports the delegation of owner
1923 or entity authorization.

1924 New authorization information is inserted by the "AuthData Insertion Protocol" (ADIP)
1925 during the creation of an entity. The "AuthData Change Protocol" (ADCP) and the
1926 "Asymmetric Authorization Change Protocol" (AACP) allow the changing of the AuthData for
1927 an entity. The protocol definitions allow expansion of protocol types to additional TCG
1928 required protocols and vendor specific protocols.

1929 The protocols use a "rolling nonce" paradigm. This requires that a nonce from one side be in
1930 use only for a message and its reply. For instance, the TPM would create a nonce and send
1931 that on a reply. The requestor would receive that nonce and then include it in the next
1932 request. The TPM would validate that the correct nonce was in the request and then create
1933 a new nonce for the reply. This mechanism is in place to prevent replay attacks and man-
1934 in-the-middle attacks.

1935 The basic protocols do not provide long-term protection of AuthData that is the hash of a
1936 password or other low-entropy entities. The TPM designer and application writer must
1937 supply additional protocols if protection of these types of data is necessary.

1938 The design criterion of the protocols is to allow for ownership authentication, command and
1939 parameter authentication and prevent replay and man-in-the-middle attacks.

1940 The passing of the AuthData, nonces and other parameters must follow specific guidelines
1941 so that commands coming from different computer architectures will interoperate properly.

1942 **End of informative comment**

1943 1. AuthData MUST use one of the following protocols

 Copyright © TCG

   a. OIAP
   b. OSAP
   c. DSAP
2. Entity creation MUST use one of the following protocols
   a. ADIP
3. Changing AuthData MUST use one of the following protocols
   a. ADCP
   b. AACP
4. The TPM MAY support additional protocols to authenticate, insert and change AuthData.
5. When a command has more than one AuthData value
   a. Each AuthData MUST use the same SHA-1 of the parameters
6. Keys MAY specify authDataUsage -> TPM_AUTH_NEVER
   a. If the caller changes the tag from TPM_TAG_RQU_AUTH1_xxx to TPM_TAG_RQU_XXX the TPM SHALL ignore the AuthData values
   b. If the caller leaves the tag as TPM_TAG_RQU_AUTH1
      i. The TPM will compute the AuthData based on the value store in the AuthData location within the key, IGNORING the state of the AuthDataUsage flag.
   c. Users may choose to use a well-known value for the AuthData when setting AuthDataUsage to TPM_AUTH_NEVER.
   d. If a key has AuthDataUsage set to TPM_AUTH_ALWAYS but is received in a command with the tag TPM_TAG_RQU_COMMAND, the command MUST return an error code.
7. For commands that normally have 2 authorization sessions, if the tag specifies only one in the parameter array, then the first session listed is ignored (authDataUsage must be TPM_AUTH_NEVER for this key) and the incoming session data is used for the second auth session in the list.
8. Keys MAY specify AuthDataUsage -> TPM_AUTH_PRIV_USE_ONLY
   a. If the key used in a command to read/access the public portion of the key (e.g. TPM_CertifyKey, TPM_GetPubKey)
      i. If the caller changes the tag from TPM_TAG_RQU_AUTH1_xxx to TPM_TAG_RQU_XXX the TPM SHALL ignore the AuthData values
      ii. If the caller leaves the tag as TPM_TAG_RQU_AUTH1
      iii. The TPM will compute the AuthData based on the value store in the AuthData location within the key, IGNORING the state of the AuthDataUsage flag
   b. else if the key used in command to read/access the private portion of the key(e.g. TPM_Sign)

1981 i. If the tag is TPM_TAG_RQU_COMMAND, the command MUST return an error
1982 code.

Copyright © TCG

1983

1984 13.1 Authorization Session Setup

1985 **Start of informative comment**

1986 The TPM provides two protocols for authorizing the use of entities without revealing the
1987 AuthData on the network or the connection to the TPM. In both cases, the protocol
1988 exchanges nonce-data so that both sides of the transaction can compute a hash using
1989 shared secrets and nonce-data. Each side generates the hash value and can compare to the
1990 value transmitted. Network listeners cannot directly infer the AuthData from the hashed
1991 objects sent over the network.

1992 The first protocol is the Object-Independent Authorization Protocol (OIAP), which allows the
1993 exchange of nonces with a specific TPM. Once an OIAP session is established, its nonces
1994 can be used to authorize the use of any entity managed by the TPM. The session can live
1995 indefinitely until either party requests the session termination. The TPM_OIAP function
1996 starts the OIAP session.

1997 The second protocol is the Object Specific Authorization Protocol (OSAP). The OSAP allows
1998 establishment of an authentication session for a single entity. The session creates nonces
1999 that can authorize multiple commands without additional session-establishment overhead,
2000 but is bound to a specific entity. The TPM_OSAP command starts the OSAP session. The
2001 TPM_OSAP specifies the entity to which the authorization is bound.

2002 Most commands allow either form of authorization protocol. In general, however, the OIAP
2003 is preferred – it is more generally useful because it allows usage of the same session to
2004 provide authorization for different entities. The OSAP is, however, necessary for operations
2005 that set or reset AuthData.

2006 OIAP sessions were designed for reasons of efficiency; only one setup process is required for
2007 potentially many authorizations.

2008 An OSAP session is doubly efficient because only one setup process is required for
2009 potentially many authorization calculations and the entity AuthData secret is required only
2010 once. This minimizes exposure of the AuthData secret and can minimize human interaction
2011 in the case where a person supplies the AuthData information. The disadvantage of the
2012 OSAP is that a distinct session needs to be setup for each entity that requires authorization.
2013 The OSAP creates an ephemeral secret that is used throughout the session instead of the
2014 entity AuthData secret. The ephemeral secret can be used to provide confidentiality for the
2015 introduction of new AuthData during the creation of new entities. Termination of the OSAP
2016 occurs in two ways. Either side can request session termination (as usual) but the TPM
2017 forces the termination of an OSAP session after use of the ephemeral secret for the
2018 introduction of new AuthData.

2019 For both the OSAP and the OIAP, session setup is independent of the commands that are
2020 authorized. In the case of OIAP, the requestor sends the TPM_OIAP command, and with the
2021 response generated by the TPM, can immediately begin authorizing object actions. The
2022 OSAP is very similar, and starts with the requestor sending a TPM_OSAP operation, naming
2023 the entity to which the authorization session should be bound.

2024 The DSAP session is to provide delegated authorization information.

All session types use a "rolling nonce" paradigm. This means that the TPM creates a new nonce value each time the TPM receives a command using the session.

Example OIAP and OSAP sessions are used to illustrate session setup and use. The fictitious command named TPM_Example occupies the place where an ordinary TPM command might be used, but does not have command specific parameters. The session connects to a key object within the TPM. The key contains AuthData that will be used to secure the session.

There could be as many as 2 authorization sessions applied to the execution of a single TPM command or as few as 0. The number of sessions used is determined by TCG 1.2 Command Specification and is indicated by the command ordinal parameter.

It is also possible to secure authorization sessions using ephemeral shared-secrets. Rather than using AuthData contained in the stored object (e.g. key), the AuthData is supplied as a parameter to OSAP session creation. In the examples below the key.usageAuth parameter is replaced by the ephemeral secret.

End of informative comment

13.2 Parameter Declarations for OIAP and OSAP Examples

Start of informative comment

To follow OIAP and OSAP protocol examples (Table 13:c and

Table 13:d), the reader should become familiar with the parameters declared in Table 13:a and Table 13:b.

Several conventions are used in the parameter tables that may facilitate readability.

The Param column (Table 13:a) identifies the sequence in which parameters are packaged into a command or response message as well as the size in bytes of the parameter value. If this entry in the row is blank, that parameter is not included in the message. <> in the size column means that the size of the element is variable. It is defined either explicitly by the preceding parameter, or implicitly by the parameter type.

The HMAC column similarly identifies the parameters that are included in HMAC calculations. This column also indicates the default parameters that are included in the audit log. Exceptions are noted under the specific ordinal, e.g. TPM_ExecuteTransport.

The HMAC # column details the parameters used in the HMAC calculation. Parameters 1S, 2S, etc. are concatenated and hashed to inParamDigest or outParamDigest, implicitly called 1H1 and possibly 1H2 if there are two authorization sessions. For the first session, 1H1, 2H1, 3H1, and 4H1 are concatenated and HMAC'ed. For the second session, 1H2, 2H2, 3H2, and 4H2 are concatenated and HMAC'ed.

In general, key handles are not included in HMAC calculations. This allows a lower software layer to map the physical handle value generated by the TPM to a logical value used by an upper software layer. The upper layer generally holds the HMAC key and generates the HMAC. Excluding the key handle allows the mapping to occur without breaking the HMAC. It is important to use a different authorization secret for each key to prevent a man-in-the-middle from altering the key handle.

The Type column identifies the TCG data type corresponding to the passed value. An encapsulation of the parameter type is not part of the command message.

Copyright © TCG

The Name column is a fictitious variable name that aids in following the examples and descriptions.

The double-lined row separator distinguishes authorization session parameters from command parameters. In Table 13:a the TPM_Example command has three parameters; keyHandle, inArgOne and inArgTwo. The tag, paramSize and ordinal parameters are message header values describing contents of a command message. The parameters below the double-lined row are OIAP / OSAP /DSAP or transport authorization session related. If a second authorization session were used, the table would show a second authorization section delineated by a second double-lined row. The authorization session parameters identify shared-secret values, session nonces, session digest and flags.

In this example, a single authorization session is used signaled by the TPM_TAG_RQU_AUTH1_COMMAND tag.

For an OIAP or transport session, the TPM_AUTHDATA description column specifies the HMAC key.

For an OSAP or DSAP session, the HMAC key is the shared secret that was calculated during the session setup, not the key specified in the description. The key specified in the description was previously used in the shared secret calculation.

2084

Param		HMAC		Type	Name	Description
#	Sz	#	Sz			
1	2			TPM_TAG	tag	TPM_TAG_RQU_AUTH1_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4	1S	4	TPM_COMMAND_CODE	ordinal	Command ordinal, fixed value of TPM_Example
4	4			TPM_KEY_HANDLE	keyHandle	Handle of a loaded key.
5	1	2S	1	BOOL	inArgOne	The first input argument
6	20	3S	20	UNIT32	inArgTwo	The second input argument.
7	4			TPM_AUTHHANDLE	authHandle	The authorization handle used for keyHandle authorization.
		2H1	20	TPM_NONCE	authLastNonceEven	Even nonce previously generated by TPM to cover inputs
8	20	3 H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
9	1	4 H1	1	BOOL	continueAuthSession	The continue use flag for the authorization handle
10	20			TPM_AUTHDATA	inAuth	The AuthData digest for inputs and keyHandle. HMAC key: key.usageAuth.

2085 **Table 13:a - Authorization Protocol Input Parameters**

Param		HMAC		Type	Name	Description
#	Sz	#	Sz			
1	2			TPM_TAG	Tag	TPM_TAG_RSP_AUTH1_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4	1S	4	TPM_RESULT	returnCode	The return code of the operation. See section 4.3.
		2S	4	TPM_COMMAND_CODE	ordinal	Command ordinal, fixed value of TPM_Example
4	4	3S	4	UINT32	outArgOne	Output argument
5	20	2 H1	20	TPM_NONCE	nonceEven	Even nonce newly generated by TPM to cover outputs
		3 H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
6	1	4 H1	1	BOOL	continueAuthSession	Continue use flag, TRUE if handle is still active
7	20			TPM_AUTHDATA	resAuth	The AuthData digest for the returned parameters. HMAC key: key.usageAuth.

2086

2087 **Table 13:b - Authorization Protocol Output Parameters**

2088 **End of informative comment**

Copyright © TCG

13.2.1 Object-Independent Authorization Protocol (OIAP)

Start of informative comment

The purpose of this section is to describe the authorization-related actions of a TPM when it receives a command that has been authorized with the OIAP protocol. OIAP uses the TPM_OIAP command to create the authorization session.

Many commands use OIAP authorization. The following description is therefore necessarily abstract. A fictitious TPM command, TPM_Example is used to represent ordinary TPM commands.

Assume that a TPM user wishes to send command TPM_Example. This is an authorized command that uses the key denoted by keyHandle. The user must know the AuthData for keyHandle (key.usageAuth) as this is the entity that requires authorization and this secret is used in the authorization calculation. Let us assume for this example that the caller of TPM_Example does not need to authorize the use of keyHandle for more than one command. This use model points to the selection of the OIAP as the authorization protocol.

For the TPM_Example command, the inAuth parameter provides the authorization to execute the command. The following table shows the commands executed, the parameters created and the wire formats of all of the information.

<inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne, inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode, ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order: authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

There are two even nonces used to execute TPM_Example, the one generated as part of the TPM_OAIP command (labeled authLastNonceEven below) and the one generated with the output arguments of TPM_Example (labeled as nonceEven below).

2114

Caller	On the wire	Dir	TPM
Send TPM_OIAP	TPM_OIAP	→	Create session Create authHandle Associate session and authHandle Generate authLastNonceEven Save authLastNonceEven with authHandle
Save authHandle, authLastNonceEven	authHandle, authLastNonceEven	←	Returns
Generate nonceOdd Compute inAuth = HMAC (key.usageAuth, inParamDigest, inAuthSetupParams) Save nonceOdd with authHandle			
Send TPM_Example	tag paramSize ordinal keyHandle inArgOne inArgTwo authHandle nonceOdd continueAuthSession inAuth	→	TPM retrieves key.usageAuth (key must have been previously loaded) Verify authHandle points to a valid session, mismatch returns TPM_E_INVALIDAUTH Retrieve authLastNonceEven from internal session storage HM = HMAC (key.usageAuth, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_E_INVALIDAUTH Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(key.usageAuth, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC(key.usageAuth, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

2115

2116 Suppose now that the TPM user wishes to send another command using the same session.
2117 For the purposes of this example, we will assume that the same example command is used
2118 (ordinal = TPM_Example). However, a different key (newKey) with its own secret
2119 (newKey.usageAuth) is to be operated on. To re-use the previous session, the
2120 continueAuthSession output boolean must be TRUE.

2121 The previous example shows the command execution reusing an existing authorization
2122 session. The parameters created and the wire formats of all of the information.

2123 In this case, authLastNonceEven is the nonceEven value returned by the TPM with the
2124 output parameters from the first protocol example.

2125

Caller	On the wire	Dir	TPM
Generate nonceOdd Compute inAuth = HMAC (newKey.usageAuth, inParamDigest, inAuthSetupParams) Save nonceOdd with authHandle			
Send TPM_Example	tag paramSize ordinal keyHandle inArgOne inArgTwo nonceOdd continueAuthSession inAuth	→	TPM retrieves newKey.usageAuth (newKey must have been previously loaded) Retrieve authLastNonceEven from internal session storage HM = HMAC (newKey.usageAuth, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_E_INVALIDAUTH Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(newKey.usageAuth, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC(newKey.usageAuth, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

2126

2127 The TPM user could then use the session for further authorization sessions. Suppose,
2128 however, that the TPM user no longer requires the authorization session. There are three
2129 possibilities in this case:

2130 The user issues a TPM_Terminate_Handle command to the TPM (section 5.3).

2131 The input argument continueAuthSession can be set to FALSE for the last command. In
2132 this case, the output continueAuthSession value will be FALSE.

2133 In some cases, the TPM automatically terminates the authorization session regardless of the
2134 input value of continueAuthSession. In this case as well, the output continueAuthSession
2135 value will be FALSE.

2136 When an authorization session is terminated for any reason, the TPM invalidates the
2137 session's handle and terminates the session's thread (releases all resources allocated to the
2138 session).

2139 **End of informative comment**

2140 OIAP Actions

2141 1. The TPM MUST verify that the authorization handle (H, say) referenced in the command
2142 points to a valid session. If it does not, the TPM returns the error code
2143 TPM_INVALID_AUTHHANDLE

2144 2. The TPM SHALL retrieve the latest version of the caller's nonce (nonceOdd) and
2145 continueAuthSession flag from the input parameter list, and store it in internal TPM
2146 memory with the authSession 'H'.

2147 3. The TPM SHALL retrieve the latest version of the TPM's nonce stored with the
2148 authorization session H (authLastNonceEven) computed during the previously executed
2149 command.

2150 4. The TPM MUST retrieve the secret AuthData (SecretE, say) of the target entity. The
2151 entity and its secret must have been previously loaded into the TPM.

2152 5. The TPM SHALL perform a HMAC calculation using the entity secret data, ordinal, input
2153 command parameters and authorization parameters according to previously specified
2154 normative regarding HMAC calculation.

2155 6. The TPM SHALL compare HM to the AuthData value received in the input parameters. If
2156 they are different, the TPM returns the error code TPM_AUTHFAIL if the authorization
2157 session is the first session of a command, or TPM_AUTH2FAIL if the authorization
2158 session is the second session of a command. Otherwise, the TPM executes the command
2159 which (for this example) produces an output that requires authentication.

2160 7. The TPM SHALL generate a nonce (nonceEven).

2161 8. The TPM creates an HMAC digest to authenticate the return code, return values and
2162 authorization parameters to the same entity secret according to previously specified
2163 normative regarding HMAC calculation.

2164 9. The TPM returns the return code, output parameters, authorization parameters and
2165 AuthData digest.

2166 10. If the output continueUse flag is FALSE, then the TPM SHALL terminate the session.
2167 Future references to H will return an error.

Copyright © TCG

2168 ### 13.2.2 Object-Specific Authorization Protocol (OSAP)

2169 **Start of informative comment**

2170 This section describes the actions of a TPM when it receives a TPM command via OSAP
2171 session. Many TPM commands may be sent to the TPM via an OSAP session. Therefore, the
2172 following description is necessarily abstract.

2173 The OSAP session is initialized through the creation of an ephemeral secret which is used to
2174 protect session traffic. Sessions are created using the TPM_OSAP command. This section
2175 illustrates OSAP using a fictitious command called TPM_Example.

2176 Assume that a TPM user wishes to send the TPM_Example command to the TPM. The
2177 keyHandle signifies that an OSAP session is being used and has the value "Auth1". The
2178 user must know the AuthData for keyHandle (key.usageAuth) as this is the entity that
2179 requires authorization and this secret is used in the authorization calculation.

2180 Let us assume that the sender needs to use this key multiple times but does not wish to
2181 obtain the key secret more than once. This might be the case if the usage AuthData were
2182 derived from a typed password. This use model points to the selection of the OSAP as the
2183 authorization protocol.

2184 For the TPM_Example command, the inAuth parameter provides the authorization to
2185 execute the command. The following table shows the commands executed, the parameters
2186 created and the wire formats of all of the information.

2187 <inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne,
2188 inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode,
2189 ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order:
2190 authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the
2191 following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

2192 In addition to the two even nonces generated by the TPM (authLastNonceEven and
2193 nonceEven) that are used for TPM_OIAP, there is a third, labeled nonceEvenOSAP that is
2194 used to generate the shared secret. For every even nonce, there is also an odd nonce
2195 generated by the system.

2196

Caller	On the wire	Dir	TPM
Send TPM_OSAP	TPM_OSAP keyHandle nonceOddOSAP	→	Create session & authHandle Generate authLastNonceEven Save authLastNonceEven with authHandle Save the ADIP encryption scheme with authHandle Generate nonceEvenOSAP Generate sharedSecret = HMAC(key.usageAuth, nonceEvenOSAP, nonceOddOSAP) Save keyHandle, sharedSecret with authHandle
Save authHandle, authLastNonceEven Generate sharedSecret = HMAC(key.usageAuth, nonceEvenOSAP, nonceOddOSAP) Save sharedSecret	authHandle, authLastNonceEven nonceEvenOSAP	←	Returns
Generate nonceOdd & save with authHandle. Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)			
Send TPM_Example	tag paramSize ordinal keyHandle inArgOne inArgTwo authHandle nonceOdd continueAuthSession inAuth	→	Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Execute TPM_Example and create returnCode. If TPM_Example requires ADIP encryption, use the algorithm indicated when the OSAP session was set up. Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

2197

2198 Table 13:c - Example OSAP Session

2199 Suppose now that the TPM user wishes to send another command using the same session
2200 to operate on the same key. For the purposes of this example, we will assume that the same
2201 ordinal is to be used (TPM_Example). To re-use the previous session, the
2202 continueAuthSession output boolean must be TRUE.

2203 The following table shows the command execution, the parameters created and the wire
2204 formats of all of the information.

2205 In this case, authLastNonceEven is the nonceEven value returned by the TPM with the
2206 output parameters from the first execution of TPM_Example.

2207

Caller	On the wire	Dir	TPM
Generate nonceOdd Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Save nonceOdd with authHandle			
Send TPM_Example	tag paramSize ordinal keyHandle inArgOne inArgTwo nonceOdd continueAuthSession inAuth	→	Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

2208

2209 Table 13:d - Example Re-used OSAP Session

2210 The TPM user could then use the session for further authorization sessions or terminate it
2211 in the ways that have been described above in TPM_OIAP. Note that termination of the
2212 OSAP session causes the TPM to destroy the shared secret.

2213 **End of informative comment**

2214 OSAP Actions

2215 1. The TPM MUST have been able to retrieve the shared secret (Shared, say) of the target
2216 entity when the authorization session was established with TPM_OSAP. The entity and
2217 its secret must have been previously loaded into the TPM.

2218 2. The TPM MUST verify that the authorization handle (H, say) referenced in the command
2219 points to a valid session. If it does not, the TPM returns the error code
2220 TPM_INVALID_AUTHHANDLE.

2221 3. The TPM MUST calculate the HMAC (HM1, say) of the command parameters according
2222 to previously specified normative regarding HMAC calculation.

2223 4. The TPM SHALL compare HM1 to the AuthData value received in the command. If they
2224 are different, the TPM returns the error code TPM_AUTHFAIL if the authorization session
2225 is the first session of a command, or TPM_AUTH2FAIL if the authorization session is the
2226 second session of a command., the TPM executes command C1 which produces an
2227 output (O, say) that requires authentication and uses a particular return code (RC, say).

2228 5. The TPM SHALL generate the latest version of the even nonce (nonceEven).

2229 6. The TPM MUST calculate the HMAC (HM2) of the return parameters according to
2230 previously specified normative regarding HMAC calculation.

2231 7. The TPM returns HM2 in the parameter list.

2232 8. The TPM SHALL retrieve the continue flag from the received command. If the flag is
2233 FALSE, the TPM SHALL terminate the session and destroy the thread associated with
2234 handle H.

2235 9. If the shared secret was used to provide confidentiality for data in the received
2236 command, the TPM SHALL terminate the session and destroy the thread associated with
2237 handle H.

2238 10. Each time that access to an entity (key) is authorized using OSAP, the TPM MUST
2239 ensure that the OSAP shared secret is that derived from the entity using TPM_OSAP.

 Copyright © TCG

13.3 Authorization Session Handles

Start of informative comment

The TPM generates authorization handles to allow for the tracking of information regarding a specific authorization invocation.

The TPM saves information specific to the authorization, such as the nonce values, ephemeral secrets and type of authentication in use.

The TPM may create any internal representation of the handle that is appropriate for the TPM's design. The requestor always uses the handle in the authorization structure to indicate authorization structure in use.

The TPM must support a minimum of two concurrent authorization handles. The use of these handles is to allow the Owner to have an authorization active in addition to an active authorization for an entity.

To ensure garbage collection and the proper removal of security information, the requestor should terminate all handles. Termination of the handle uses the continue-use flag to indicate to the TPM that the handle should be terminated.

Termination of a handle instructs the TPM to perform garbage collection on all AuthData. Garbage collection includes the deletion of the ephemeral secret.

End of informative comment

1. The TPM MUST support authorization handles. See Section 23 Session pool.
2. The TPM MUST support authorization-handle termination. The termination includes secure deletion of all authorization session information.

13.4 Authorization-Data Insertion Protocol (ADIP)

Start of informative comment

The ADIP allows for the creation of new entities and the secure insertion of the new entity AuthData. The transmission of the new AuthData uses encryption with the key based on the shared secret of an OSAP session.

The creation of AuthData is the responsibility of the entity owner. He or she may use whatever process he or she wishes. The transmission of the AuthData from the entity owner to the TPM requires confidentiality and integrity. These requirements assume the insertion of the AuthData occurs over a network. While local insertion of the data would not require these measures, the protocol is established to be consistent with both local and remote insertions. The confidentiality of the transmission comes from the encryption of the AuthData. The integrity comes from the OSAP session HMAC.

When the requestor is sending the AuthData to the TPM, the command requires the authorization of the entity parent. For example, to create a new TPM identity key and set its AuthData requires the AuthData of the TPM Owner. To create a new wrapped key requires the AuthData of the parent key.

The creation of a new entity requires the authorization of the entity owner. When the requestor starts the creation process, the creator must establish an OSAP session using the parent of the new entity.

For the mandatory XOR encryption algorithm, the creator builds an encryption key using a SHA-1 hash of the OSAP shared secret and a session nonce. The creator XOR encrypts the new AuthData using the encryption key as a one-time pad and sends this encrypted data along with the creation request to the TPM. The TPM decrypts the AuthData using the same OSAP shared secret and session nonce.

The XOR encryption algorithm is sufficient for almost all use models. There may be additional use models where a different encryption algorithm would be beneficial. The TPM may support AES as an additional encryption algorithm. The key and IV or counter use the OSAP shared secret and session nonces.

The creator believes that the OSAP creates a shared secret known only to the creator and the TPM. The TPM believes that the creator is the entity owner by their knowledge of the parent entity AuthData. The creator believes that the process completed correctly and that the AuthData is correct because the HMAC will only verify with the OSAP shared secret.

In the following example, we want to send the previously described command TPM_EXAMPLE to create a new entity. In the example, we assume there is a third input parameter encAuth, and that one of the input parameters is named parentHandle to reference the parent for the new entity (e.g., the SRK and its children).

Copyright © TCG

2297

Caller	On the wire	Dir	TPM
Send TPM_OSAP	TPM_OSAP parentHandle nonceOddOSAP	→	Create session & authHandle Generate authLastNonceEven Save authLastNonceEven with authHandle Save the ADIP encryption scheme with authHandle Generate nonceEvenOSAP Generate sharedSecret = HMAC(parent.usageAuth, nonceEvenOSAP, nonceOddOSAP) Save parentHandle, sharedSecret with authHandle
Save authHandle, authLastNonceEven Generate sharedSecret = HMAC(parent.usageAuth, nonceEvenOSAP, nonceOddOSAP) Save sharedSecret	authHandle, authLastNonceEven nonceEvenOSAP	←	Returns
Generate nonceOdd & save with authHandle. Compute input parameter newAuth = XOR(entityAuthData, SHA1(sharedSecret, authLastNonceEven)) Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)			
Send TPM_Example	tag paramSize ordinal parentHandle inArgOne inArgTwo encAuth authHandle nonceOdd continueAuthSession inAuth	→	Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Execute TPM_Example: decrypt encAuth to entityAuth, create entity and build returnCode. Use the ADIP encryption scheme indicated when the OSAP session was set up. Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters Terminate the authorization session associated with authHandle

2298

2299 Table 13:e - Example ADIP Session

2300 **End of informative comment**

2301 1. The TPM MUST enable ADIP by using the OSAP or DSAP

2302 a. The upper byte of the entity type indicates the encryption scheme.

2303 b. The TPM internally stores the encryption scheme as part of the session and enforces
2304 the encryption choice on the subsequent use of the session.

2305 c. When TPM_ENTITY_TYPE is used for ordinals other than TPM_OSAP or TPM_DSAP
2306 (i.e., for cases where there is no ADIP encryption action), the TPM_ENTITY_TYPE
2307 upper byte MUST be 0x00.

2308 2. The TPM MUST destroy the session whenever a new entity AuthData is created.

2309 3. The TPM MUST encrypt the AuthData for the new entity.

2310 a. The TPM MUST support the XOR encryption scheme.

2311 b. The TPM MAY support AES symmetric key encryption schemes.

2312 i. If TPM_PERMANENT_FLAGS -> FIPS is TRUE

2313 (1) All encrypted authorizations MUST use a symmetric key encryption scheme.

2314 a. Encrypted AuthData values occur in the following commands

2315 i. TPM_CreateWrapKey

2316 ii. TPM_ChangeAuth

2317 iii. TPM_ChangeAuthOwner

2318 iv. TPM_Seal

2319 v. TPM_Sealx

2320 vi. TPM_MakeIdentity

2321 vii. TPM_CreateCounter

2322 viii. TPM_CMK_CreateKey

2323 ix. TPM_NV_DefineSpace

2324 (1) This ordinal contains a special case where no encryption is used.

2325 x. TPM_Delegate_CreateKeyDelegation

2326 xi. TPM_Delegate_CreateOwnerDelegation

2327 4. If the entity type indicates XOR encryption for the AuthData secret

2328 a. Create X1 the SHA-1 of the concatenation of (authHandle -> sharedSecret ||
2329 authLastNonceEven).

2330 b. Create the decrypted AuthData the XOR of X1 and the encrypted AuthData.

2331 c. If the command ordinal contains a second AuthData2 secret (e.g.
2332 TPM_CreateWrapKey)

2333 i. Create X2 the SHA-1 of the concatenation of (authHandle -> sharedSecret ||
2334 nonceOdd).

2335 ii. Create the decrypted AuthData2 the XOR of X2 and the encrypted AuthData2.

2336 5. If the entity type indicates symmetric key encryption

 Copyright © TCG

a. The key for the encryption algorithm is the first bytes of the OSAP shared secret.
   i. E.g., For AES128, the key is the first 16 bytes of the OSAP shared secret.
   ii. There is no support for AES keys greater than 128 bits.
b. If the entity type indicates CTR mode
   i. The initial counter value for AuthData is the first bytes of authLastNonceEven.
      (1) E.g., For AES128, the initial counter value is the first 16 bytes of authLastNonceEven.
   ii. If the command ordinal contains a second AuthData2 secret (e.g. TPM_CreateWrapKey)
      (1) The initial counter value for AuthData2 is the first bytes of nonceOdd.
   iii. Additional counter values as required are generated by incrementing the counter value as described in 31.1.3 TPM_ES_SYM_CTR.

Start of informative comment

The method of incrementing the counter value is different from that used by some standard crypto libraries (e.g. openSSL, Java JCE) that increment the entire counter value. TPM users should be aware of this to avoid errors when the counter wraps.

End of informative comment

13.5 AuthData Change Protocol (ADCP)

Start of informative comment

All entities from the Owner to the SRK to individual keys and data blobs have AuthData. This data may need to change at some point in time after the entity creation. The ADCP allows the entity owner to change the AuthData. The entity owner of a wrapped key is the owner of the parent key.

A requirement is that the owner must remember the old AuthData. The only mechanism to change the AuthData when the entity owner forgets the current value is to delete the entity and then recreate it.

To protect the data from exposure to eavesdroppers or other attackers, the AuthData uses the same encryption mechanism in use during the ADIP.

Changing AuthData requires opening two authentication handles. The first handle authenticates the entity owner (or parent) and the right to load the entity. This first handle is an OSAP and supplies the data to encrypt the new AuthData according to the ADIP protocol. The second handle can be either an OIAP or an OSAP, it authorizes access to the entity for which the AuthData is to be changed.

The AuthData in use to generate the OSAP shared secret must be the AuthData of the parent of the entity to which the change will be made.

When changing the AuthData for the SRK, the first handle OSAP must be setup using the TPM Owner AuthData. This is because the SRK does not have a parent, per se.

If the SRKAuth data is known to userA and userB, userA can snoop on userB while userB is changing the AuthData for a child of the SRK, and deduce the child's newAuth. Therefore, if SRKAuth is a well known value, TPM_ChangeAuthAsymStart and TPM_ChangeAuthAsymFinish are preferred over TPM_ChangeAuth when changing AuthData for children of the SRK.

This applies to all children of the SRK, including TPM identities.

End of informative comment

1. Changing AuthData for the TPM SHALL require authorization of the current TPM Owner.
2. Changing AuthData for the SRK SHALL require authorization of the TPM Owner.
3. If SRKAuth is a well known value, TPM_ChangeAuth SHOULD NOT be used to change the AuthData value of a child of the SRK, including the TPM identities.
4. All other entities SHALL require authorization of the parent entity.

Copyright © TCG

13.6 Asymmetric Authorization Change Protocol (AACP)

Start of informative comment

This is now deprecated. Use the normal change session inside of a transport session with confidentiality.

This asymmetric change protocol allows the entity owner to change entity authorization, under the parent's execution authorization, to a value of which the parent has no knowledge.

In contrast, the TPM_ChangeAuth command uses the parent entity AuthData to create the shared secret that encrypts the new AuthData for an entity. This creates a situation where the parent entity ALWAYS knows the AuthData for entities in the tree below the parent. There may be instances where this knowledge is not a good policy.

This asymmetric change process requires two commands and the use of an authorization session.

End of informative comment

1. Changing AuthData for the SRK SHALL involve authorization by the TPM Owner.
2. If SRKAuth is a well known value,
    a. TPM_ChangeAuthAsymStart and TPM_ChangeAuthAsymFinish SHOULD be used to change the AuthData value of a child of the SRK, including the TPM identities.
3. All other entities SHALL involve authorization of the parent entity.

2407 # 14. FIPS 140 Physical Protection

2408 **Start of informative comment**

2409 The FIPS 140-2 program provides assurance that a cryptographic device performs properly.
2410 It is appropriate for TPM vendors to attempt to obtain FIPS 140-2 certification.

2411 The TPM design should be such that the TPM vendor has the opportunity of obtaining FIPS
2412 140-2 certification.

2413 **End of informative comment**

2414 ## 14.1 TPM Profile for FIPS Certification

2415 **Start of informative comment**

2416 The FIPS mode of the TPM does require some changes over the normal TPM. These changes
2417 are listed here such that there is a central point of determining the necessary FIPS changes.

2418 **Key creation and use**

2419 TPM_LoadKey, TPM_CMK_CreateKey and TPM_CreateWrapKey changed to disallow the
2420 creation or loading of TPM_AUTH_NEVER, legacy and keys less than 1024 bits.
2421 TPM_MakeIdentity changed to disallow TPM_AUTH_NEVER.

2422 **End of informative comment**

2423 1. Each TPM Protected Capability MUST be designed such that some profile of the
2424 Capability is capable of obtaining FIPS 140-2 certification

Copyright © TCG

2425 # 15. Maintenance

2426 **Start of informative comment**

2427 The maintenance feature is a vendor-specific feature, and its implementation is vendor-
2428 specific. The implementation must, however, meet the minimum security requirements so
2429 that implementations of the maintenance feature do not result in security weaknesses.

2430 There is no requirement that the maintenance feature is available, but if it is implemented,
2431 then the requirements must be met.

2432 The maintenance feature described in the specification is an example only, and not the only
2433 mechanism that a manufacturer could implement that meets these requirements.

2434 Maintenance is different from backup/migration, because maintenance provides for the
2435 migration of both migratory and non-migratory data. Maintenance is an optional TPM
2436 function, but if a TPM enables maintenance, the maintenance capabilities in this
2437 specification are mandatory – no other migration capabilities shall be used. Maintenance
2438 necessarily involves the manufacturer of a Subsystem.

2439 When maintaining computer systems, it is sometimes the case that a manufacturer or its
2440 representative needs to replace a Subsystem containing a TPM. Some manufacturers
2441 consider it a requirement that there be a means of doing this replacement without the loss
2442 of the non-migrational keys held by the original TPM.

2443 The owner and users of TCG platforms need assurance that the data within protected
2444 storage is adequately protected against interception by third parties or the manufacturer.

2445 This process MUST only be performed between two platforms of the same manufacturer and
2446 model. If the maintenance feature is supported, this section defines the required functions
2447 defined at a high level. The final function definitions and entire maintenance process is left
2448 to the manufacturer to define within the constraints of these high level functions.

2449 Any maintenance process must have certain properties. Specifically, any migration to a
2450 replacement Subsystem must require collaboration between the Owner of the existing
2451 Subsystem and the manufacturer of the existing Subsystem. Further, the procedure must
2452 have adequate safeguards to prevent a non-migrational key being transferred to multiple
2453 Subsystems.

2454 The maintenance capabilities TPM_CreateMaintenanceArchive and
2455 TPM_LoadMaintenanceArchive enable the transfer of all Protected Storage data from a
2456 Subsystem containing a first TPM (TPM_1) to a Subsystem containing a second TPM (TPM_2):

2457 A manufacturer places a public key in non-volatile storage into its TPMs at manufacture
2458 time.

2459 The Owner of TPM_1 uses TPM_CreateMaintenanceArchive to create a maintenance archive
2460 that enables the migration of all data held in Protected Storage by TPM_1. The Owner of TPM_1
2461 must provide his or her authorization to the Subsystem. The TPM then creates the
2462 TPM_MIGRATE_ASYMKEY structure and follows the process defined.

2463 The XOR process prevents the manufacturer from ever obtaining plaintext TPM_1 data.

2464 The additional random data provides a means to assure that a maintenance process cannot
2465 subvert archive data and hide such subversion.

2466 The random mask can be generated by two methods, either using the TPM RNG or MGF1 on
2467 the TPM Owners AuthData.

2468 The manufacturer takes the maintenance blob, decrypts it with its private key, and satisfies
2469 itself that the data bundle represents data from that Subsystem manufactured by that
2470 manufacturer. Then the manufacturer checks the endorsement certificate of TPM_2 and
2471 verifies that it represents a platform to which data from TPM_1 may be moved.

2472 The manufacturer dispatches two messages.

2473 The first message is made available to CAs, and is a revocation of the TPM_1 endorsement
2474 certificate.

2475 The second message is sent to the Owner of TPM_2, which will communicate the SRK,
2476 tpmProof and the manufacturer's permission to install the maintenance blob only on TPM_2

2477 The Owner uses TPM_LoadMaintenanceArchive to install the archive copy into TPM_2, and
2478 overwrite the existing TPM_2-SRK and TPM_2-tpmProof in TPM_2. TPM_2 overwrites TPM_2-SRK
2479 with TPM_1-SRK, and overwrites TPM_2-tpmProof with TPM_1-tpmProof.

2480 Note that the command TPM_KillMaintenanceFeature prevents the operation of
2481 TPM_CreateMaintenanceArchive and TPM_LoadMaintenanceArchive. This enables an Owner
2482 to block maintenance (and hence the migration of non-migratory data) either to or from a
2483 TPM.

2484 It is required that a manufacturer takes steps that prevent further access of migrated data
2485 by $TPM_{1.}$ This may be achieved by deleting the existing Owner from TPM_1, for example.

2486 For the manufacturer to validate that the maintenance blob is coming from a valid TPM, the
2487 manufacturer can require that a TPM identity sign the maintenance blob. The identity
2488 would be from a CA under the control of the manufacturer and hence the manufacturer
2489 would be satisfied that the blob is from a valid TPM.

2490 **End of informative comment**

2491 1. The maintenance feature MUST ensure that the information can be on only one TPM at
2492 a time. Maintenance MUST ensure that at no time the process will expose a shielded
2493 location. Maintenance MUST require the active participation of the Owner.

2494 2. Any migration of non-migratory data protected by a Subsystem SHALL require the
2495 cooperation of both the Owner of that non-migratory data and the manufacturer of that
2496 Subsystem. That manufacturer SHALL NOT cooperate in a maintenance process unless
2497 the manufacturer is satisfied that non-migratory data will exist in exactly one
2498 Subsystem. A TPM SHALL NOT provide capabilities that support migration of non-
2499 migratory data unless those capabilities are described in the TCG specification.

2500 3. The maintenance feature MUST move the following

2501 4. TPM_KEY for SRK. The maintenance process will reset the SRK AuthData to match the
2502 TPM Owners AuthData

2503 5. TPM_PERMANENT_DATA -> tpmProof

2504 6. TPM Owner's authorization

2505 15.1 Field Upgrade

2506 **Start of informative comment**

Copyright © TCG

2507 A TPM, once in the field, may need to update the protected capabilities. This command,
2508 which is optional, provides the mechanism to perform the update.

2509 **End of informative comment**

2510 The TPM SHOULD have provisions for upgrading the subsystem after shipment from the
2511 manufacturer. If provided the mechanism MUST implement the following guidelines:

2512 1. The upgrade mechanisms in the TPM MUST not require the TPM to hold a global secret.
2513 The definition of global secret is a secret value shared by more than one TPM.

2514 2. The TPM is not allowed to pre-store or use unique identifiers in the TPM for the purpose
2515 of field upgrade. The TPM MUST NOT use the endorsement key for identification or
2516 encryption in the upgrade process. The upgrade process MAY use a TPM Identity (AIK) to
2517 deliver upgrade information to specific TPM devices.

2518 3. The upgrade process can only change protected-capabilities.

2519 4. The upgrade process can only access data in shielded-locations where this data is
2520 necessary to validate the TPM Owner, validate the TPME and manipulate the blob

2521 5. The TPM MUST conform to the TCG specification, protection profiles and security targets
2522 after the upgrade. The upgrade MAY NOT decrease the security values from the original
2523 security target.

2524 6. The security target used to evaluate this TPM MUST include this command in the TOE.

2525 # 16. Proof of Locality

2526 **Start of informative comment**

2527 When a platform is designed with a trusted process, the trusted process may wish to
2528 communicate with the TPM and indicate that the command is coming from the trusted
2529 process. The definition of a trusted process is a platform specific issue.

2530 The commands that the trusted process sends to the TPM are the normal TPM commands
2531 with a modifier that indicates that the trusted process initiated the command. The TPM
2532 accepts the command as coming from the trusted process merely because the modifier is
2533 set. The TPM itself is not responsible for how the signal is asserted; only that it honors the
2534 assertions. The TPM cannot verify the validity of the modifier.

2535 The definition of the modifier is a platform specific issue. Depending on the platform, the
2536 modifier could be a special bus cycle or additional input pins on the TPM. The assumption
2537 is that spoofing the modifier to the TPM requires more than just a simple hardware attack,
2538 but would require expertise and possibly special hardware. One example would be special
2539 cycles on the LPC bus that inform the TPM it is under the control of a process on the PC
2540 platform.

2541 To allow for multiple mechanisms and for finer grained reporting, the TPM will include 4
2542 locality modifiers. These four modifiers allow the platform specific specification to properly
2543 indicate exactly what is occurring and for TPM's to properly respond to locality.

2544 **End of informative comment**

2545 1. The TPM modifies the receipt of a command and indicates that the trusted process sent
2546 the command when the TPM determines that the modifier is on. The modifier MUST only
2547 affect the individual command just received and MUST NOT affect any other commands.
2548 However, TPM_ExecuteTransport MUST propagate the modifier to the wrapped
2549 command.

2550 2. A TPM platform specific specification MAY indicate the presence of a maximum of 4 local
2551 modifiers. The modifier indication uses the TPM_MODIFIER_INDICATOR data type.

2552 3. The received modifier MUST indicate a single level.

2553 4. The definition of the trusted source is in the platform specific specification.

2554 5. For ease in reading this specification the indication that the TPM has received any
2555 modifier will be LOCAL_MOD = TRUE.

 Copyright © TCG

2556 # 17. Monotonic Counter

2557 **Start of informative comment**

2558 The monotonic counter provides an ever-increasing incremental value. The TPM must
2559 support at least 4 concurrent counters. Implementations inside the TPM may create 4
2560 unique counters or there may be one counter with pointers to keep track of the pointers
2561 current value. A naming convention to allow for unambiguous reference to the various
2562 components the following terms are in use:

2563 Internal Base – This is the main counter. It is in use internally by the TPM and is not
2564 directly accessible by any outside process.

2565 External Counter – A counter in use by external processes. This could be related to the
2566 main counter via pointers and difference values or it could be a totally unique value. The
2567 value of an external counter is not affected by any use, increment or deletion of any other
2568 external counter.

2569 Max Value – The max count value of all counters (internal and external). So if there were 3
2570 external counters having values of 10, 15 and 201 and the internal base having a value of
2571 201 then Max Value is 201. In the same example if the internal base was 502 then Max
2572 Value would be 502.

2573 There are two methods of obtaining an external count, signed or unsigned. The external
2574 counter must allow for 7 years of increments every 5 seconds without causing a hardware
2575 failure. The output of the counter is a 32-bit value.

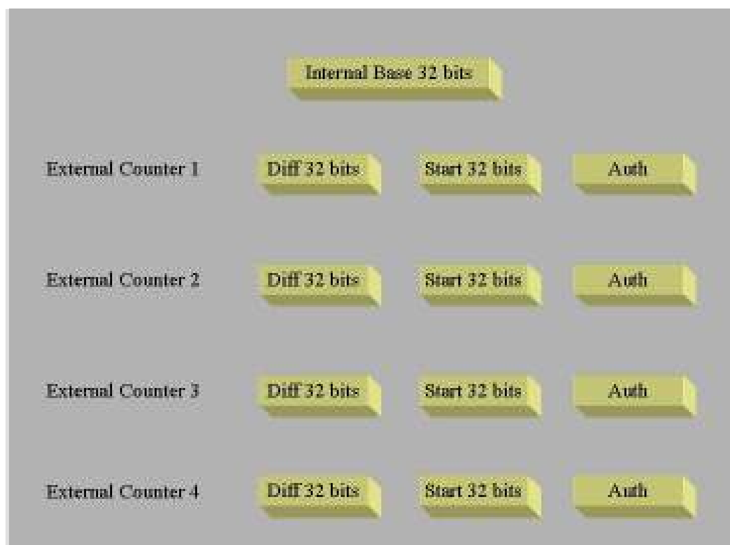
2576 The TPM may create a throttling mechanism that limits the ability to increment an external
2577 counter within a certain time range. The TPM must support an increment rate of once every
2578 5 seconds.

2579 To create an external counter requires TPM Owner authorization. To increment an external
2580 counter the command must pass authorization to use the counter.

2581 External counters can be tagged with a short text string to facilitate counter administration.

2582 Manufacturers are free to implement the monotonic counter using any mechanism.

2583 To illustrate the counters and base the following example is in use. This mechanism uses
2584 two saving values (diff and start), however this is only an example and not meant to indicate
2585 any specific implementation.



The internal base (IB) always moves forward and can never be reset. IB drives all external counters on the machine.

The purpose of the following example is to show the two external counters always moving forward independent of the other and how the IB moves forward also.

Starting condition is that IB is at 22 and no other external counters are active.

Start external counter A

Increment IB (set new Max Value) IB = 23

Assign start value of A to 23 (or Max Value)

Assign difference of A to 23 (we always start at current value of IB)

Assign a handle for A

Increment A 5 times

IB is now 28

Request current A value

Return 28 = 28 (IB) + 23 (difference) – 23 (start value)

Counter A has gone from the start of 23 to 28 incremented 5 times.

TPM_Startup(ST_CLEAR)

Start Counter B

Save A difference 28 = 23 (old difference) + 28 (IB) – 23 (start value)

Increment IB (set new Max Value) IB = 29

Set start value of B to 29 (or Max Value)

Assign difference of B to 29

Assign handle for B

Increment B 8 times

IB is now 37

2611 Request B value

2612 Return 37 = 37 (IB) + 29 (difference) – 29 (start value)

2613 TPM_Startup(ST_CLEAR)

2614 Increment A

2615 Store B difference (37)

2616 Load A start value of 37

2617 Increment IB to 38

2618 Return A value

2619 Return 29 = 38 (IB) + 28 (difference) – 37 (start value)

2620

2621 Notice that A has gone from 28 to 29 which is correct, while B is at 37. Depending on the
2622 order of increments A may pass B or it may always be less than B.

2623 **End of informative comment**

2624 1. The counter MUST be designed to not wear out in the first 7 years of operation. The
2625 counter MUST be able to increment at least once every 5 seconds. The TPM, in response
2626 to operations that would violate these counter requirements, MAY throttle the counter
2627 usage (cause a delay in the use of the counter) or return the error
2628 TPM_E_COUNTERUSAGE.

2629 2. The TPM MUST support at least 4 concurrent counters.

2630 3. The establishment of a new counter MUST prevent the reuse of any previous counter
2631 value. I.E. if the TPM has 3 counters and the max value of a current counter is at 36
2632 then the establishment of a new counter would start at 37.

2633 4. After a successful TPM_Startup(ST_CLEAR) the first successful TPM_IncrementCounter
2634 sets the counter handle. Any attempt to issue TPM_IncrementCounter with a different
2635 handle MUST fail.

2636 5. TPM_CreateCounter does NOT set the counter handle.

2637 # 18. Transport Protection

2638 **Start of informative comment**

2639 The creation of sessions allows for the grouping of a set of commands into a session. The
2640 session provides a log of all commands and can provide confidentiality of the commands
2641 using the session.

2642 Session establishment creates a shared secret and then uses the shared secret to authorize
2643 and protect commands sent to the TPM using the session.

2644 After establishing the session, the caller uses the session to wrap a command to execute.
2645 The user of the transport session can wrap any command except for commands that would
2646 create nested transport sessions.

2647 The log of executed commands uses a structure that includes the parameters and current
2648 tick count. The session log provides a record of each command using the session.

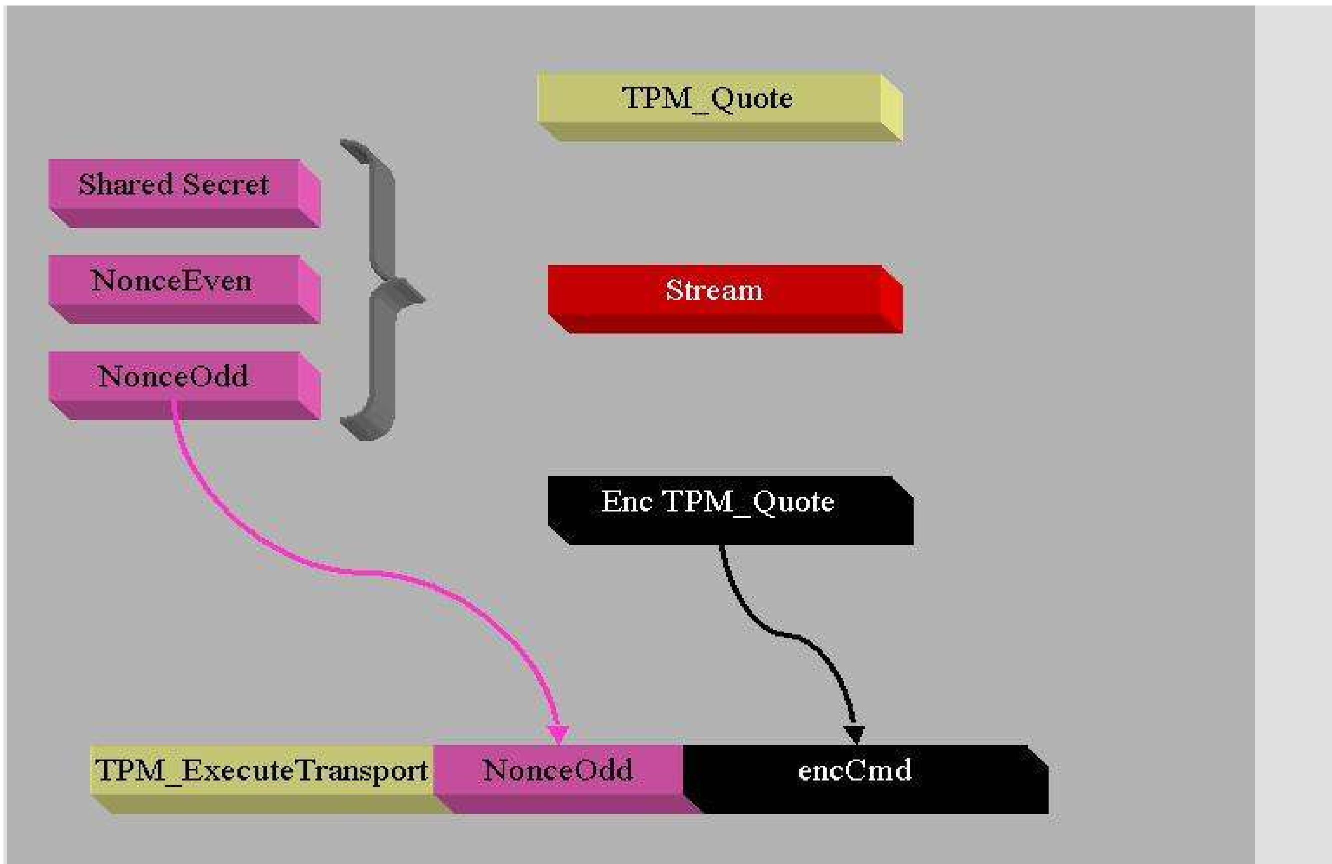
2649 The transport session uses the same rolling nonce protocol that authorization sessions use.
2650 This protocol defines two nonces for each command sent to the TPM; nonceOdd provided by
2651 the caller and nonceEven generated by the TPM.

2652 For confidentiality, the caller can use the MGF1 function to create an XOR string the same
2653 size as the command to execute. The inputs to the MGF1 function are the shared secret,
2654 nonceOdd and nonceEven. A symmetric key encryption algorithm can also be specified.

2655 There is no explicit close session as the caller can use the continueSession flag set to false
2656 to end a session. The caller can also call the sign session log, which also ends the session. If
2657 the caller losses track of which sessions are active the caller should use the flush
2658 commands to regain control of the TPM resources.

2659 For an attacker to successfully break the encryption the attacker must be able to determine
2660 from a few bits what an entire SHA-1 output was. This is equivalent to breaking SHA-1. The
2661 reason that the attacker will know some bits is that the commands are in a known format.
2662 This then allows the attacker to determine what the XOR bits were. Knowledge of 159 bits of
2663 the XOR stream does not provide any greater that 50% probability of knowing the 160th bit.

 Copyright © TCG



This picture shows the protection of a TPM_Quote command. Previously executed was session establishment. The nonces in use for the TPM_Quote have no relationship with the nonces that are in use for the TPM_ExecuteTransport command.

End of informative comment

1. The TPM MUST support a minimum of one transport session.
2. The TPM MUST NOT support the nesting of transport sessions. The definition of nesting is attempting to execute a wrapped command that is a transport session command. So for example when executing TPM_ExecuteTransport the wrapped command MUST not be TPM_ExecuteTransport.
3. The TPM MUST ensure that if transport logging is active that the inclusion of the tick count in the session log does not provide information that would make a timing attack on the operations using the session more successful.
4. The transport session MAY be exclusive. Any command executed outside of the exclusive transport session MUST cause the invalidation of the exclusive transport session.
    a. The TPM_ExecuteTransport command specifying the exclusive transport session is the only command that does not terminate the exclusive session.
5. It MAY be ineffective to wrap TPM_SaveState in a transport session. Since the TPM MAY include transport sessions in the saved state, the saved state MAY be invalidated by the wrapping TPM_ExecuteTransport.

2684 ## 18.1 Transport encryption and authorization

2685 **Start of informative comment**

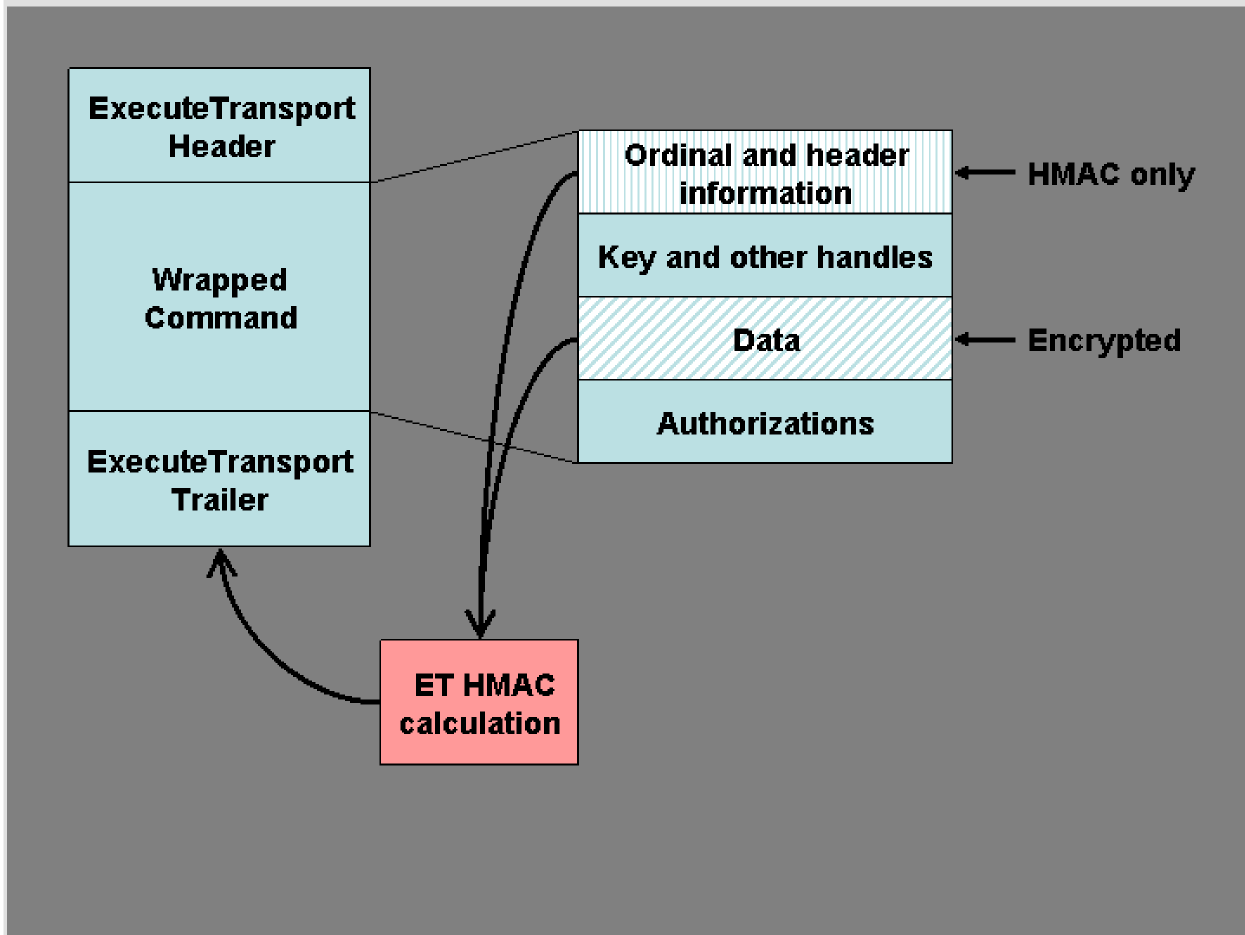
2686 The confidentially of the transport protection is provided by a encrypting the wrapped
2687 command. Encryption of various items in the wrapped command makes resource
2688 management of a TPM impossible. For this reason, encryption of the entire command is not
2689 possible. In addition to the encryption issue, there are difficulties with creating the HMAC
2690 for the TPM_ExecuteTransport authorization.

2691 The solution to these problems is to provide limited encryption and HMAC information.

2692 The HMAC will only include two areas from the wrapped command, the command header
2693 information up to the handles, and the data after the handles. The format of all TPM
2694 commands is such that all handles are in the data stream prior to the payload or data. After
2695 the data comes the authorization information. To enable resource management, the HMAC
2696 for TPM_ExecuteTransport only includes the ordinal, header information and the data. The
2697 HMAC does not include handles and the authorization handles and nonces.

2698 The exception is TPM_OwnerReadInternalPub, which uses fixed value key handles that are
2699 included in the encryption and HMAC calculation.

2700



2701

2702 A more exact representation of the execute transport command would be the following

```
2703 **********************************************
2704 * TAGet | LENet | ORDet | wrappedCmd | AUTHet *
2705 **********************************************
2706
```

2707 wrappedCmd looks like

```
2708 ******************************************************************
2709 * TAGw | LENw | ORDw | HANDLESw | DATAw | AUTH1w (o) | AUTH2w (o) *
2710 ******************************************************************
```

2711 A more exact representation of the execute transport response would be the following

```
2712 *********************************************
2713 * TAGet | LENet | RCet | wrappedRsp | AUTHet *
2714 *********************************************
2715
```

2716 wrappedRsp looks like

```
2717 *****************************************************************
2718 * TAGw | LENw | RCw | HANDLESw | DATAw | AUTH1w (o) | AUTH2w (o) *
2719 *****************************************************************
2720
```

2721 The calculation for AUTHet takes as the data component of the HMAC calculation the
2722 concatenation of ORDw and DATAw. A normal HMAC calculation would have taken the
2723 entire wrappedCmd value but for the executeTransport calculation only the above two
2724 values are active. This does require the executeTransport command to parse the
2725 wrappedCmd to find the appropriate values.

2726 The data for the command HMAC calculation is the following:

2727 H1 = SHA-1 (ORDw || DATAw)

2728 inParamDigest = SHA-1 (ORDet || wrappedCmdSize || H1)

2729 AUTHet = HMAC (inParamDigest || lastNonceEven(et) || nonceOdd(et) || continue(et))

2730 The data for the response HMAC calculation is the following:

2731 H2 = SHA-1 (RCw || ORDw || DATAw)

2732 outParamDigest = SHA-1 (RCet || ORDet || currentTicks || locality || wrappedRspSize ||
2733 H1)

2734 AUTHet = HMAC (outParamDigest || nonceEven(et) || nonceOdd(et) || continue(et))

2735 DATAw is the unencrypted data. wrappedCmdSize and wrappedRspSize ares the actual size
2736 of the DATAw area and not the size of H1 or H2.

2737 **End of informative comment**

2738 The TPM MUST release a transport session and all information related to the session when:

2739 1. TPM_ReleaseTransportSigned is executed

2740 2. TPM_ExecuteTransport is executed with continueTransSession set to FALSE

2741 3. Any failure of the integrity check during execution of TPM_ExecuteTransport

4. If the session has TPM_TRANSPORT_LOG set and the TPM tick session is interrupted for any reason. This is due to the return of tick values without the nonces associated with the session.
5. The TPM executes some command that deactivates the TPM or removes the TPM Owner or EK.

18.1.1 MGF1 parameters

Start of informative comment

MGF1 provides the confidentiality for the transport session. MGF1 is a function from PKCS 1 version 2.0. This function provides a mechanism to distribute entropy over a large sequence. The sequence provides a value to XOR over the message. This in effect creates a stream cipher but not one that is available for bulk encryption.

Transport confidentiality uses MGF1 as a stream cipher and obtains the entropy for each message from the following three parameters; nonceOdd, nonceEven and session authData.

It is imperative that the stream cipher not use the same XOR sequence at any time. The following illustrates how the sequence changes for each message (both input and output).

M1Input – N2, N1, sessionSecret)

M1Output – N4, N1, sessionSecret)

M2Input – N4, N3, sessionSecret)

M2Output – N6, N3, sessionSecret)

There is an issue with this sequence. If the caller does not change N1 to N3 between M1Output and M2Input then the same sequence will be generated. The TPM does not enforce the requirement to change this value so it is possible to leak information.

The fix for this is to add one more parameter, the direction. So the sequence is now this:

M1Input – N2, N1, “in”, sessionSecret)

M1Output – N4, N1, “out”, sessionSecret)

M2Input – N4, N3, “in”, sessionSecret)

M2Output – N6, N3, “out”, sessionSecret)

Where “in” indicates the in direction and “out” indicates the out direction.

Notice the calculation for M1Output uses “out” and M2Input uses “in”, so if the caller makes a mistake and does not change nonceOdd, the sequence will still be different.

nonceEven is under control of the TPM and is always changing, so there is no need to worry about nonceEven not changing.

End of informative comment

18.1.2 HMAC calculation

Start of informative comment

The HMAC calculation for transports presents some issues with what should and should not be in the calculation. The idea is to create a calculation for the wrapped command and add that to the wrapper.

 Copyright © TCG

2780 So the data area for a wrapped command is not entirely HMAC'd like a normal command
2781 would be.

2782 The process is to calculate the inParamDigest of the unencrypted wrapped command
2783 according to the normal rules of command HMAC calculations. Then use that value as the
2784 3S parameter in the calculation. 2S is the actual wrapped command size, and not the size
2785 of inParamDigest.

2786 Example using a wrapped TPM_LoadKey command

2787 Calculate the SHA-1 value for the TPM_LoadKey command (ordinal and data) as per the
2788 normal HMAC rules. Take the digest and use that value as 3S for the
2789 TPM_ExecuteTransport HMAC calculation.

2790 **End of informative comment**

2791 18.1.3 Transport log creation

2792 **Start of informative comment**

2793 The log of information that a transport session creates needs a mechanism to tie any keys
2794 in use during the session to the session. As the HMAC and encryption for the command
2795 specifically exclude handles, there is no direct way to create the binding.

2796 When creating the transport input log, if the handle(s) points to a key or keys, the public
2797 keys are digested into the log. The session owner knows the value of any keys in use and
2798 hence can still create a log that shows the values used by the log and can validate the
2799 session.

2800 **End of informative comment**

2801 18.1.4 Additional Encryption Mechanisms

2802 **Start of informative comment**

2803 The TPM can optionally implement alternate algorithms for the encryption of commands
2804 sent to the TPM_ExecuteTransport command. The designation of the algorithm uses the
2805 TPM_ALGORITHM_ID and TPM_ENC_SCHEME elements of the TPM_TRANSPORT_PUBLIC
2806 parameter of the TPM_EstablishTransport command.

2807 The anticipation is that AES will be supported by various TPM's. Symmetric algorithms
2808 have options available to them like key size, block size and operating mode. When using an
2809 algorithm other than MGF1 the algorithm and scheme must specify these options.

2810 **End of informative comment**

2811 1. The TPM MAY support other symmetric algorithms for the confidentiality requirement in
2812 TPM_EstablishTransport

2813 18.2 Transport Error Handling

2814 **Start of informative comment**

2815 With the transport hiding the actual execution of commands and the transport capable of
2816 generating errors, rules must be established to allow for the errors and the results of
2817 commands to be properly passed to TPM callers.

End of informative comment

1. There are 3 error cases:
2. C1 is the case where an error occurs during the processing of the transport package at the TPM. In this case, the wrapped command has not been sent to the command decoder. Errors occurring during C1 are sent back to the caller as a response to the TPM_ExecuteTransport command. The error response does not have confidentiality.
3. C2 is the case where an error occurs during the processing of the wrapped command. This results in an error response from the command. The session returns the error response according to the attributes of the session.
4. C3 is the case where an error occurs after the wrapped command has completed processing and the TPM is preparing the response to the TPM_ExecuteTransport command. In this case, where the TPM does have an internal error, the TPM has no choice but to return the error as in C1. This however hides the results of the wrapped command. If the wrapped command completed successfully then there are session nonces that are being returned to the caller that are lost. The loss of these nonces causes the caller to be unsure of the state of the TPM and requires the reestablishment of sessions and keys.

18.3 Exclusive Transport Sessions

Start of informative comment

The caller may establish an exclusive session with the TPM. When an exclusive session is running, execution of any command other then TPM_ExecuteTransport or TPM_ReleaseTransportSigned targeting the exclusive session causes the abnormal invalidation of the exclusive transport session. Invalidation means that the handle is no longer valid and all subsequent attempts to use the handle return an error.

The design for the exclusive session provides an assurance that no other command executed on the TPM. It is not a lock to prevent other operations from occurring. Therefore, the caller is responsible for ensuring no interruption of the sequence of commands using the TPM.

One exclusive session

The TPM only supports one exclusive session at a time. There is no nesting or other commands possible. The TPM maintains an internal flag that indicates the existence of an exclusive session.

TSS responsibilities

It is the responsibility of the TSS (or other controlling software) to ensure that only commands using the session reach the TPM. As the purpose of the session is to show that nothing else occurred on the TPM during the session, the TSS should control access to the TPM and prevent any other uses of the TPM. The TSS design must take into account the possibility of exclusive session handle invalidation.

Sleep states

Exclusive sessions as defined here do not work across TPM_SaveState and TPM_Startup(ST_STATE) invocations. To have this sequence work properly there would need to be exceptions to allowing only TPM_ExecuteTranport and

Copyright © TCG

2860 TPM_ReleaseTransportSigned in an exclusive session. The requirement for these exceptions
2861 would come from the attempt of the TSS to understand the current state of the TPM.
2862 Commands like TPM_GetCapability and others would have to execute to inform the TSS as
2863 to the internal state of the TPM. For this reason, there are no exceptions to the rule and the
2864 exclusive session does not remain active across a TPM_SaveState command.

2865 **End of informative comment**

2866 1. The TPM MUST support only one exclusive transport session

2867 2. The TPM MUST invalidate the exclusive transport session upon the receipt of any
2868 command other than TPM_ExecuteTransport or TPM_ReleaseTransportSigned targeting
2869 the exclusive session.

2870 a. Invalidation includes the release of any resources assigned to the session

2871 18.4 Transport Audit Handling

2872 **Start of informative comment**

2873 Auditing of TPM_ExecuteTransport occurs as any other command that may require
2874 auditing. There are two entries in the log, one for input one for output. The execution of the
2875 wrapped command can create an anomaly in the log.

2876 Assume that both TPM_ExecuteTransport and the wrapped commands require auditing, the
2877 audit flow would look like the following:

2878 TPM_ExecuteTransport input parameters

2879 wrapped command input parameters

2880 wrapped command output parameters

2881 TPM_ExecuteTransport output parameters

2882 **End of informative comment**

2883 1. Audit failures are reported using the AUTHFAIL error commands and reflect the success
2884 or failure of the wrapped command.

2885 18.4.1 Auditing of wrapped commands

2886 **Start of informative comment**

2887 Auditing provides information to allow an auditor to recreate the operations performed.
2888 Confidentiality on the transport channel is to hide what operations occur. These two
2889 features are in conflict. According to the TPM design philosophy, the TPM Owner takes
2890 precedence.

2891 For a command sent on a transport session, with the session using confidentiality and the
2892 command requiring auditing, the TPM will execute the command however the input and
2893 output parameters for the command are ignored.

2894 **End of informative comment**

2895 1. When the wrapped command requires auditing and the transport session specifies
2896 encryption, the TPM MUST perform the audit. However, when computing the audit
2897 digest:

2898 a. For input, only the ordinal is audited.

2899 b. For output, only the ordinal and return code are audited.

Copyright © TCG

19. Audit Commands

Start of informative comment

To allow the TPM Owner the ability to determine that certain operations on the TPM have been executed, auditing of commands is possible. The audit value is a digest held internally to the TPM and externally as a log of all audited commands. With the log held externally to the TPM, the internal digest must allow the log auditor to determine the presence of attacks against the log. The evidence of tampering may not provide evidence of the type of attack mounted against the log.

The TPM cannot enforce any protections on the external log. It is the responsibility of the external log owner to properly maintain and protect the log.

The TPM provides mechanisms for the external log maintainer to resynchronize the internal digest and external logs.

The Owner has the ability to set which functions generate an audit event and to change which functions generate the event at any time.

The status of the audit generation is not sensitive information and so the command to determine the status of the audit generation is not an owner authorized command.

It is important to note the difference between auditing and the logging of transport sessions. The audit log provides information on the execution of specific commands. There will be a very limited number of audited commands, most likely those commands that provide identities and control of the TPM. Commands such as TPM_Unseal would not be audited. They would use the logging functions of a transport session.

The auditing of an ordinal happens in a two-step process. The first step involves auditing the receipt of the command and the input parameters; the second step involves auditing the response to the command and the output parameters. This two-step process is in place to lower the amount of memory necessary to keep track of the audit while executing the command. This two-step process makes no memory requirements on a TPM to save any audit information while a command is executing.

There is a requirement to enable verification of the external audit log both during a power session and across power sessions and to enable detection of partial or inconsistent audit logs throughout the lifetime of a TPM.

A TPM will hold an internal record consisting of a non-volatile counter (that increments once per session, when the first audit event of that session occurs) and a digest (that holds the digest of the current session). Most probably, the audit digest will be volatile. Note, however, that nothing in this specification prevents the use of a non-volatile audit digest. This arrangement of counter and digest is advantageous because it is easier to build a high endurance non-volatile counter than a high endurance non-volatile digest. This arrangement is insufficient, however, because the truncation of an audit log of any session is possible without trace. It is therefore necessary to perform an explicit close on the audit session. If there is no record of a close-audit event in an audit session, anything could have happened after the last audit event in the audit log. The essence of a typical TPM audit recording mechanism is therefore:

The TPM contains a volatile digest used like a PCR, where the “integrity metrics” are digests of command parameters in the current audit session.

2943 An audit session opens when the volatile "PCR" digest is "extended" from its NULL state.
2944 This occurs whenever an audited command is executed AND no audit session currently
2945 exists, and in no other circumstances. When an audit session opens, a non-volatile counter
2946 is automatically incremented.

2947 An audit session closes when a TPM receives TPM_GetAuditDigestSigned with a closeAudit
2948 parameter asserted. An audit session must be considered closed if the value in the volatile
2949 digest is invalid (for whatever reason).

2950 TPM_GetCapability should report the effect of TPM_Startup on the volatile digest. (TPMs
2951 may initialize the volatile digest on the first audit command after TPM_Startup(ST_CLEAR),
2952 or on the first audit command after any version of TPM_Startup, or may be independent of
2953 TPM_Startup.)

2954 When the TPM signs its audit digest, it signs the concatenation of the non-volatile counter
2955 and the volatile digest, and exports the value of the non-volatile counter, plus the value of
2956 the volatile digest, plus the value of the signature.

2957 If the audit digest is initialized by TPM_Startup(ST_STATE), then it may be useless to audit
2958 the TPM_SaveState ordinal. Any command after TPM_SaveState MAY invalidate the saved
2959 state. If authorization sessions are part of the saved state, TPM_GetAuditDigestSigned will
2960 most likely invalidate the state as it changes the preserved authorization session nonce. It
2961 may therefore be impossible to get the audit results.

2962 The system designer needs to ensure that the selected TPM can handle the specific
2963 environment and avoid burnout of the audit monotonic counter.

2964 **End of informative comment**

2965 1. Audit functionality is optional

2966 a. If the platform specific specification requires auditing, the specification SHALL
2967 indicate how the TPM implements audit

2968 2. The TPM MUST maintain an audit monotonic count that is only available for audit
2969 purposes.

2970 a. The increment of this audit counter is under the sole control of the TPM and is not
2971 usable for other count purposes.

2972 b. This monotonic count MUST BE incremented by one whenever the audit digest is
2973 "extended" from a NULL state.

2974 3. The TPM MUST maintain an audit digest.

2975 a. This digest MUST be set to all zeros upon the execution of
2976 TPM_GetAuditDigestSigned with a TRUE value of closeAudit provided that the
2977 signing key is an identity key.

2978 b. This digest MAY be set to all zeros on TPM_Startup[ST_CLEAR] or
2979 TPM_Startup[ST_STATE].

2980 c. When an audited command is executed, this register MUST be extended with the
2981 digest of that command.

2982 4. Each command ordinal has an indicator in non-volatile TPM memory that indicates if
2983 execution of the command will generate an audit event. The setting of the ordinal
2984 indicator MUST be under control of the TPM Owner.

Copyright © TCG

5. Updating of auditDigest MAY cease when TPM_STCLEAR_FLAGS -> deactivated is TRUE. This is because a deactivated TPM performs no useful service until the TPM_Startup(ST_CLEAR), at which point TPM_STCLEAR_FLAGS -> deactivated is reinitialized.

19.1 Audit Monotonic Counter

Start of informative comment

The audit monotonic counter (AMC) performs the task of sequencing audit logs across audit sessions. The AMC must have no other uses other than the audit log.

The TPM and platform should be matched such that the expected AMC endurance matches the expected platform audit sessions and sleep cycles.

Given the size of the AMC it is not anticipated that the AMC would roll over. If the AMC were to roll over, and the storage of the AMC still allowed updates, the AMC could cycle and start at 0 again.

End of informative comment

1. The AMC is a TPM_COUNTER_VALUE.
2. The AMC MUST last for 7 years or at least 1,000,000 audit sessions, whichever occurs first. After this amount of usage, there is no guarantee that the TPM will continue to properly increment the monotonic counter.

3003 # 20. Design Section on Time Stamping

3004 **Start of informative comment**

3005 The TPM provides a service to apply a time stamp to various blobs. The time stamp provided
3006 by the TPM is not an actual universal time clock (UTC) value but is the number of timer
3007 ticks the TPM has counted. It is the responsibility of the caller to associate the ticks to an
3008 actual UTC time.

3009 The TPM counts ticks from the start of a timing session. Timing sessions are platform
3010 dependent events that may or may not coincide with TPM_Init and TPM_Startup sessions.
3011 The reason for this difference is the availability of power to the TPM. In a PC desktop, for
3012 instance power could be continually available to the TPM by using power from the wall
3013 socket. For a PC mobile platform, power may not be available when only using the internal
3014 battery. It is a platform designer's decision as to when and how they supply power to the
3015 TPM to maintain the timing ticks.

3016 The TPM can provide a time stamping service. The TPM does not maintain an internal
3017 secure source of time rather the TPM maintains a count of the number of ticks that have
3018 occurred since the start of a timing session.

3019 On a PC, the TPM may use the timing source of the LPC bus or it may have a separate clock
3020 circuit. The anticipation is that availability of the TPM timing ticks and the tick resolution is
3021 an area of differentiation available to TPM manufactures and platform providers.

3022 **End of informative comment**

3023 1. This specification makes no requirement on the mechanism required to implement the
3024 tick counter in the TPM.

3025 2. This specification makes no requirement on the ability for the TPM to maintain the
3026 ability to increment the tick counter across power cycles or in different power modes on
3027 a platform.

3028 ## 20.1 Tick Components

3029 **Start of informative comment**

3030 The TPM maintains for each tick session the following values:

3031 Tick Count Value (TCV) – The count of ticks for the session.

3032 Tick Increment Rate (TIR) – The rate at which the TCV is incremented. There is a set
3033 relationship between TIR and seconds, the relationship is set during manufacturing of the
3034 TPM and platform. This is the TPM_CURRENT_TICKS -> tickRate parameter.

3035 Tick Session Nonce (TSN) – The session nonce is set at the start of each tick session.

3036 **End of informative comment**

3037 1. The TCV MUST be set to 0 at the start of each tick session. The TPM MUST start a new
3038 tick session if the TPM loses the ability to increment the TCV according to the TIR.

3039 2. The TSN MUST be set to the next value from the TPM RNG at the start of each new tick
3040 session. When the TPM loses the ability to increment the TCV according to the TIR the
3041 TSN MUST be set to all zeros.

 Copyright © TCG

3. If the TPM discovers tampering with the tick count (through timing changes etc) the TPM MUST treat this as an attack and shut down further TPM processing as if a self-test had failed.

20.2 Basic Tick Stamp

Start of informative comment

The TPM does not provide a secure time source, nor does it provide a signature over some time value. The TPM does provide a signature over some current tick counter. The signature covers a hash of the blob to stamp, the current counter value, the tick session nonce and some fixed text.

The Tick Stamp Result (TSR) is the result of the tick stamp operation that associates the TCV, TSN and the blob. There is no association with the TCV or TSR with any UTC value at this point.

End of informative comment

20.3 Associating a TCV with UTC

Start of informative comment

An outside observer would like to associate a TCV with a relevant time value. The following shows how to accomplish this task. This protocol is not required but shows how to accomplish the job.

EntityA wants to have BlobA time stamped. EntityA performs TPM_TickStamp on BlobA. This creates TSRB (TickStampResult for Blob). TSRB records TSRBTCV, the current value of the TCV, and associates TSRBTCV with the TSN.

Now EntityA needs to associate a TCV with a real time value. EntityA creates blob TS which contains some known text like “Tick Stamp”. EntityA performs TPM_TickStamp on blob TS creating TSR1. This records TSR1TCV, the current value of the TCV, and associates TSR1TCV with the TSN.

EntityA sends TSR1 to a Time Authority (TA). TA creates TA1 which associates TSR1 with UTC1.

EntityA now performs TPM_TickStamp on TA1. This creates TSR2. TSR2 records TSR2TCV, the current values of the TCV, and associates TSR2TCV with the TSN.

Analyzing the associations

EntityA has three TSR’s; TSRB the TSR of the blob that we wanted to time stamp, TSR1 the TSR associated with the TS blob and TSR2 the TSR associated with the information from the TA. EntityA wants to show an association between the various TSR such that there is a connection between the UTC and BlobA.

From TSR1 EntityA knows that TSR1TCV is less than the UTC. This is true since the TA is signing TSR1 and the creation of TSR1 has to occur before the signature of TSR1. Stated mathematically:

$$TSR1TCV < UTC1$$

From TSR2 EntityA knows that TSR2TCV is greater than the UTC. This is true since the TPM is signing TA1 which must be created before it was signed. Stated mathematically:

TSR2TCV > UTC1

EntityA now knows TSR1TCV and TSR2TCV bound UTC1. Stated mathematically:

TSR1TCV < UTC1 < TSR2TCV

This association holds true if the TSN for TSR1 matches the TSN for TSR2. If some event occurs that causes the TPM to create a new TSN and restart the TCV then EntityA must start the process all over again.

EntityA does not know when UTC1 occurred in the interval between TSR1TCV and TSR2TCV. In fact, the value TSR2TCV minus TSR1TCV (TSRDELTA) is the amount of uncertainty to which a TCV value should be associated with UTC1. Stated mathematically:

TSRDELTA = TSR2TCV – TSR1TCV iff TSR1TSN = TSR2TSN

EntityA can obtains k1 the relationship between ticks and seconds using the TPM_GetCapability command. EntityA also obtains k2 the possible errors per tick. EntityA now calculate DeltaTime which is the conversion of ticks to seconds and the TSRDELTA. State mathematically:

DeltaTime = (k1 * TSRDELTA) + (k2 * TSRDELTA)

To make the association between DeltaTime, UTC and TSRB note the following:

DeltaTime = (k1*TSRDelta) + Drift = TimeChange + Drift

Where ABSOLUTEVALUE(Drift)<k2*TSRDelta

(1) TSR1TCV < UTC1 < TSR2TCV

True since you cannot sign something before it exists

(2) TSR1TCV < UTC1 < TSR1TCV + TSR2TCV-TSR1TCV <= TSR1TCV + DeltaTime (= TSR1TCV +TimeChange +Drift)

True because TSR1 and TSR2 are in the same tick session proved by the same TSN. (Note TimeChange is positive!)

(3) 0 < UTC1-TSR1TCV < DeltaTime

(Subtract TSR1TCV from all sides)

(4) 0 > TSR1TCV - UTC1 > -DeltaTime = -TimeChange - Drift

(Multiply through by -1)

(5) TimeChange/2 > [TSR1TCV - (UTC1-TimeChange/2)] > -TimeChange/2 - Drift

(add TimeChange/2 to all sides)

(6) TimeChange/2 + ABSOLUTEVALUE(Drift) > [TSR1TCV - (UTC1-TimeChange/2)]

> -TimeChange/2 - ABSOLUTEVALUE(Drift)

Making the large side of an equality bigger, and potentially making the small side smaller.

(7) ABSOLUTEVALUE[TSR1TCV - (UTC1-TimeChange/2)] < TimeChange/2 +

ABSOLUTEVALUE(Drift)

(Definition of Absolute Value, and TimeChange is positive)

From which we see that TSR1TCV is approximately UTC1-TimeChange/2 with a symmetric possible error of TimeChange/2 + AbsoluteValue(Drift)

We can calculate this error as being less than k1*TSRDelta/2 + k2*TSRDelta.

EntityA now has the ability to associate UTC1 with TSBTSV and by allow others to know that BlobA was signed at a certain time. First TSBTSN must equal TSR1TSN. This relationship allows EntityA to assert that TSRB occurs during the same session as TSR1 and TSR2.

EntityA calculates HashTimeDelta which is the difference between TSR1TCV and TSRBTCV and the conversion of ticks to seconds. HashTimeDelta includes the same k1 and k2 as calculated above. Stated mathematically:

E = k2(TSR1TCV – TSRBTCV)

HashTimeDelta = k1(TSR1TCV – TSRBTCV) + E

Now the following relationships hold:

(1) UTC1 – DeltaTime < TSRBTCV – (TSRBTCV – TSR1TCV) < UTC1

(2) UTC1 – DeltaTime < TSRBTCV + HashTimeDelta + E < UTC1

(3) UTC1 – HashTimeDelta – DeltaTime – E < TSRBTCV < UTC1 – HashTimeDelta + E

(4) TSRBTCV = (UTC1 – HashTimeDelta – DeltaTime/2) + (E + DeltaTime/2)

This has the correct properties

As DeltaTime grows so does the error bar (or the uncertainty of the time association)

As the difference between the time of the measurement and the time of the time stamp grows, so does the E as a function of E is HashTimeDelta

End of informative comment

20.4 Additional Comments and Questions

Start of informative comment

Time Difference

If two things are time stamped, say at TCVs and TCVe (for TCV at start, TCV at end) then any entity can calculate the time difference between the two events and will get:

TimeDiff = k1*|TCVe – TCVs| + k2*|TCVe – TCVs|

This TimeDiff does not indicate what time the two events occurred at it merely gives the time between the events. This time difference doesn’t require a Time Authority.

Why is TSN (tick session nonce) required?

Without it, there is no way to associate a Time Authority stamp with any TSV, as the TSV resets at the start of every tick session. The TSN proves that the concatenation of TSV and TSN is unique.

How does the protocol prevent replay attacks?

The TPM signs the TSR sent to the TA. This TSR contains the unique combination of TSV and TSN. Since the TSN is unique to a tick session and the TSV continues to increment any attempt to recreate the same TSR will fail. If the TPM is reset such that the TSV is at the same value, the TSN will be a new value. If the TPM is not reset then the TSV continues to increment and will not repeat.

How does EntityA know that the TSR1 that the TA signs is recent?

It doesn't. EntityA checks however to ensure that the TSN is the same in all TSR. This ensures that the values are all related. If TSR1 is an old value then the HashTimeDelta will be a large value and the uncertainty of the relation of the signing to the UTC will be large.

Why does associating a UTC time with a TSV take two steps?

This is because it takes some time between when a request goes to a time authority and when the response comes. The protocol measures this time and uses it to create the time deltas. The relationship of TSV to UTC is somewhere between the request and response.

Affect of power on the tick counter

As the TPM is not required to maintain an internal clock and battery, how the platform provides power to the TPM affects the ability to maintain the tick counter. The original mechanism had the TPM maintaining an indication of how the platform provided the power. Previous performance does not predict what might occur in the future, as the platform may be unable to continue to provide the power (dead battery, pulled plug from wall etc). With the knowledge that the TPM cannot accurately report the future, the specification deleted tick type from the TPM.

The information relative to what the platform is doing to provide power to the TPM is now a responsibility of the TSS. The TSS should first determine how the platform was built, using the platform credential. The TSS should also attempt to determine the actual performance of the TPM in regards to maintaining the tick count. The TSS can help in this determination by keeping track of the tick nonce. The tick nonce changes each time the tick count is lost. By comparing the tick nonce across system events the TSS can obtain a heuristic that represents how the platform provides power to the TPM.

The TSS must define a standard set of values as to when the tick nonce continues to increment across system events.

The following are some PC implementations that give the flavor of what is possible regarding the clock on a specific platform.

TICK_INC - No TPM power battery. Clock comes from PCI clock, may stop from time to time due to clock stopping protocols such as CLKRUN.

TICK_POWER - No TPM power battery. Clock source comes from PCI clock, always runs except in S3+.

TICK_STSTATE - External power (might be battery) consumed by TPM during S3 only. Clock source comes either from a system clock that runs during S3 or from crystal/internal TPM source.

TICK_STCLEAR - Standby power used to drive counter. In desktop, may be related to when system is plugged into wall. Clock source comes either from a system clock that runs when standby power is available or from crystal/internal TPM source.

Copyright © TCG

3198 TICK_ALWAYS - TPM power battery. Clock source comes either from a battery powered
3199 system clock that crystal/internal TPM source.

3200 **End of informative comment**

21. Context Management

Start of informative comment

The TPM is a device that contains limited resources. Caching of the resources may occur without knowledge or assistance from the application that loaded the resource. In version 1.1 there were two types of resources that had need of this support keys and authorization sessions. Each type had a separate load and restore operation. In version 1.2 there is the addition of transport sessions. To handle these situations generically 1.2 is defining a single context manager that all types of resources may use.

The concept is simple, a resource manager requests that wrapping of a resource in a manner that securely protects the resource and only allows the restoring of the resource on the same TPM and during the same operational cycle.

Consider a key successfully loaded on the TPM. The parent keys that loaded the key may have required a different set of PCR registers than are currently set on the TPM. For example, the end result is to have key5 loaded. Key3 is protected by key2, which is protected by key1, which is protected by the SRK. Key1 requires PCR1 to be in a certain state, key2 requires PCR2 to load and key3 requires PCR3. Now at some point in time after key1 loaded key2, PCR1 was extended with additional information. If key3 is evicted then there is no way to reload key3 until the platform is rebooted. To avoid this type of problem the TPM can execute context management routines. The context management routines save key3 in its current state and allow the TPM to restore the state without having to use the parent keys (key1 and key2).

There are numerous issues with performing context management on sessions. These issues revolve around the use of the nonces in the session. If an attacker can successfully store, attack, fail and then reload the session the attacker can repeat the attack many times.

The key that the TPM uses to encrypt blobs may be a volatile or non-volatile key. One mechanism would be for the TPM to generate a new key on each TPM_Startup command. Another would be for the TPM to generate the key and store it persistently in the TPM_PERMANENT_DATA area.

The symmetric key should be relatively the same strength as a 2048-bit RSA key. 128-bit AES would be appropriate.

End of informative comment

1. Context management is a required function.
2. Execution of the context commands MUST NOT cause the exposure of any TPM shielded location.
3. The TPM MUST NOT allow the context saving of the EK or the SRK.
4. The TPM MAY use either symmetric or asymmetric encryption. For asymmetric encryption the TPM MUST use a 2048 RSA key.
5. A wrapped session blob MUST only be loadable once. A wrapped key blob MAY be reloadable.
6. The TPM MUST support a minimum of 16 concurrent saved contexts other than keys. There is no minimum or maximum number of concurrent saved key contexts.

 Copyright © TCG

3242 7. All external session blobs (of type TPM_RT_TRANS or TPM_RT_AUTH) can be invalidated
3243 upon specific request (via TPM_FlushXXX using TPM_RT_CONTEXT as resource type),
3244 this does not include session blobs of type TPM_RT_KEY.

3245 8. External session blobs are invalidated on TPM_Startup(ST_CLEAR) or on
3246 TPM_Startup(any) based on the startup effects settings

3247 a. Session blobs of type TPM_RT_KEY with the attributes of parentPCRStatus = FALSE
3248 and isVolatile = FALSE SHOULD not invalidated on TPM_Startup(any)

3249 9. All external session invalidate automatically upon installation of a new owner due to the
3250 setting of a new tpmProof.

3251 10. If the TPM enters failure mode ALL session blobs (including keys) MUST be invalidated

3252 a. Invalidation includes ensuring that contextNonceKey and contextNonceSession will
3253 change when the TPM recovers from the failure.

3254 11. Attempts to restore a wrapped blob after the successful completion of
3255 TPM_Startup(ST_CLEAR) MUST fail. The exception is a wrapped key blob which may be
3256 long-term and which MAY restore after a TPM_Startup(ST_CLEAR).

3257 12. The save and load context commands are the generic equivalent to the context
3258 commands in 1.1. Version 1.2 deprecates the following commands:

3259 a. TPM_AuthSaveContext

3260 b. TPM_AuthLoadContext

3261 c. TPM_KeySaveContext

3262 d. TPM_KeyLoadContext

3263 # 22. Eviction

3264 **Start of informative comment**

3265 The TPM has numerous resources held inside of the TPM that may need eviction. The need
3266 for eviction occurs when the number or resources in use by the TPM exceed the available
3267 space. For resources that are hard to reload (i.e. keys tied to PCR values) the outside entity
3268 should first perform a context save before evicting items.

3269 In version 1.1 there were separate commands to evict separate resource types. This new
3270 command set uses the resource types defined for context saving and creates a generic
3271 command that will evict all resource types.

3272 **End of informative comment**

3273 1. The TPM MUST NOT flush the EK or SRK using this command.

3274 2. Version 1.2 deprecates the following commands:

3275 a. TPM_Terminate_Handle

3276 b. TPM_EvictKey

3277 c. TPM_Reset

Copyright © TCG

3278
23. Session pool

3279 **Start of informative comment**

3280 The TPM supports two types of sessions that use the rolling nonce protocol, authorization
3281 and transport. These sessions require much of the same handling and internal storage by
3282 the TPM. To allow more flexibility the internal storage for these sessions will be defined as
3283 coming from the same pool (or area).

3284 The pool requires that three (3) sessions be available. The entities using the TPM can
3285 determine the usage models of what sessions are active. This allows a TPM to have 3
3286 authorization sessions or 3 transport sessions at one time.

3287 Using all available pool resources for transport sessions is not a very usable model. If all
3288 resources are in use by transport there is no resources available for authorization sessions
3289 and hence no ability to execute any commands requiring authorization. A more realistic
3290 model would be to have two transport sessions and one authorization session. While this is
3291 an unrealistic model for actual execution there will be no requirement that the TPM prevent
3292 this from happening. A model of how it could occur would be when there are two
3293 applications running, both using 2 transport sessions and one authorization session. When
3294 switching between the applications if the requirement was that only 2 transport sessions
3295 could be active the TSS that would provide the context switch would have to ensure that the
3296 transport sessions were context saved first.

3297 Sessions can be virtualized, so while the TPM may only have 3 loaded sessions, there may
3298 be an unlimited number of context saved sessions stored outside the TPM.

3299 **End of informative comment**

3300 1. The TPM MUST support a minimum of three (3) concurrent sessions. The sessions MAY
3301 be any mix of authentication and transport sessions.

24. Initialization Operations

Start of informative comment

Initialization is the process where the TPM establishes an operating environment from a no power state. Initialization occurs in many different flavors with PCR, keys, handles, sessions and context blobs all initialized, reloaded or unloaded according to the rules and platform environment.

Initialization does not affect the operational characteristics of the TPM (like TPM Ownership).

Clear is the process of returning the TPM to factory defaults. The clear commands need protection from unauthorized use and must allow for the possibility of changing Owners. The clear process requires authorization to execute and locks to prevent unauthorized operation.

The clear functionality performs the following tasks:

Invalidate SRK. Invalidating the SRK invalidates all protected storage areas below the SRK in the hierarchy. The areas below are not destroyed they just have no mechanism to be loaded anymore.

All TPM volatile and non-volatile data is set to default value except the endorsement key pair. The clear includes the Owner-AuthData, so after performing the clear, the TPM has no Owner. The PCR values are undefined after a clear operation.

The TPM shall return TPM_NOSRK until an Owner is set. After the execution of the clear command, the TPM must go through a power cycle to properly set the PCR values.

The Owner has ultimate control of when a clear occurs.

The Owner can perform the TPM_OwnerClear command using the TPM Owner authorization. If the Owner wishes to disable this clear command and require physical access to perform the clear, the Owner can issue the TPM_DisableOwnerClear command.

During the TPM startup processing anyone with physical access to the machine can issue the TPM_ForceClear command. This command performs the clear. The TPM_DisableForceClear disables the TPM_ForceClear command for the duration of the power cycle. TSS startup code that does not issue the TPM_DisableForceClear leaves the TPM vulnerable to a denial of service attack. The assumption is that the TSS startup code will issue the TPM_DisableForceClear on each power cycle after the TSS determines that it will not be necessary to issue the TPM_ForceClear command. The purpose of the TPM_ForceClear command is to recover from the state where the Owner has lost or forgotten the TPM Ownership token.

The TPM_ForceClear must only be possible when the issuer has physical access to the platform. The manufacturer of a platform determines the exact definition of physical access.

End of informative comment

1. The TPM MUST support proper initialization. Initialization MUST properly configure the TPM to execute in the platform environment.
2. Initialization MUST ensure that handles, keys, sessions, context blobs and PCR are properly initialized, reloaded or invalidated according to the platform environment.

Copyright © TCG

3343 3. The description of the platform environment arrives at the TPM in a combination of
3344 TPM_Init and TPM_Startup.

25. HMAC digest rules

Start of informative comment

The order of calculation of the HMAC is critical to being able to validate the authorization and parameters of a command. All commands use the same order and format for the calculation.

A more exact representation of a command would be the following

```
*****************************************************************
* TAG  | LEN  | ORD  | HANDLES | DATA  | AUTH1 (o) | AUTH2  (o) *
*****************************************************************
```

The text area for the HMAC calculation would be the concatenation of the following:

ORD || DATA

End of informative comment

The HMAC digest of parameters uses the following order

1. Skip tag and length
2. Include ordinal. This is the 1S parameter in the HMAC column for each command
3. Skip handle(s). This includes key and other session handles
4. Include data and other parameters for the command. This starts with the 2S parameter in the HMAC column for each command.
5. Skip all AuthData values.

Copyright © TCG

26. Generic authorization session termination rules

Start of informative comment

These rules are the generic rules that govern all authorization sessions, a specific session type may have additional rules or modifications of the generic rules

End of informative comment

1. A TPM SHALL unilaterally perform the actions of TPM_FlushSpecific for a session upon any of the following events
   a. "continueUse" flag in the authorization session is FALSE
   b. Shared secret of the session in use to create the exclusive-or for confidentiality of data. Example is TPM_ChangeAuth terminates the authorization session. TPM_ExecuteTransport does not terminate the session due to protections inherent in transport sessions.
   c. When the associated entity is invalidated
   d. When the command returns a fatal error. This is due to error returns not setting a nonceEven. Without a new nonceEven the rolling nonces sequence is broken hence the TPM MUST terminate the session.
   e. Failure of an authorization check at the start of the command
   f. Execution of TPM_Startup(ST_CLEAR)
2. The TPM MAY perform the actions of TPM_FlushSpecific for a session upon the following events
   a. Execution of TPM_Startup(ST_STATE)

3385 # 27. PCR Grand Unification Theory

3386 **Start of informative comment**

3387 This section discusses the unification of PCR definition and use with locality.

3388 The PCR allow the definition of a platform configuration. With the addition of locality, the
3389 meaning of a configuration is somewhat larger. This section defines how the two combine to
3390 provide the TPM user information relative to the platform configuration.

3391 These are the issues regarding PCR and locality at this time

3392 **Definition of configuration**

3393 A configuration is the combination of PCR, PCR attributes and the locality.

3394 **Passing the creators configuration to the user of data**

3395 For many reasons, from the creator's viewpoint and the user's viewpoint, the configuration
3396 in use by the creator is important information. This information needs transmitting to the
3397 user with the data and with integrity.

3398 The configuration must include the locality and may not be the same configuration that will
3399 use the data. This allows one configuration to seal a value for future use and the end user
3400 to know the genealogy of where the data comes from.

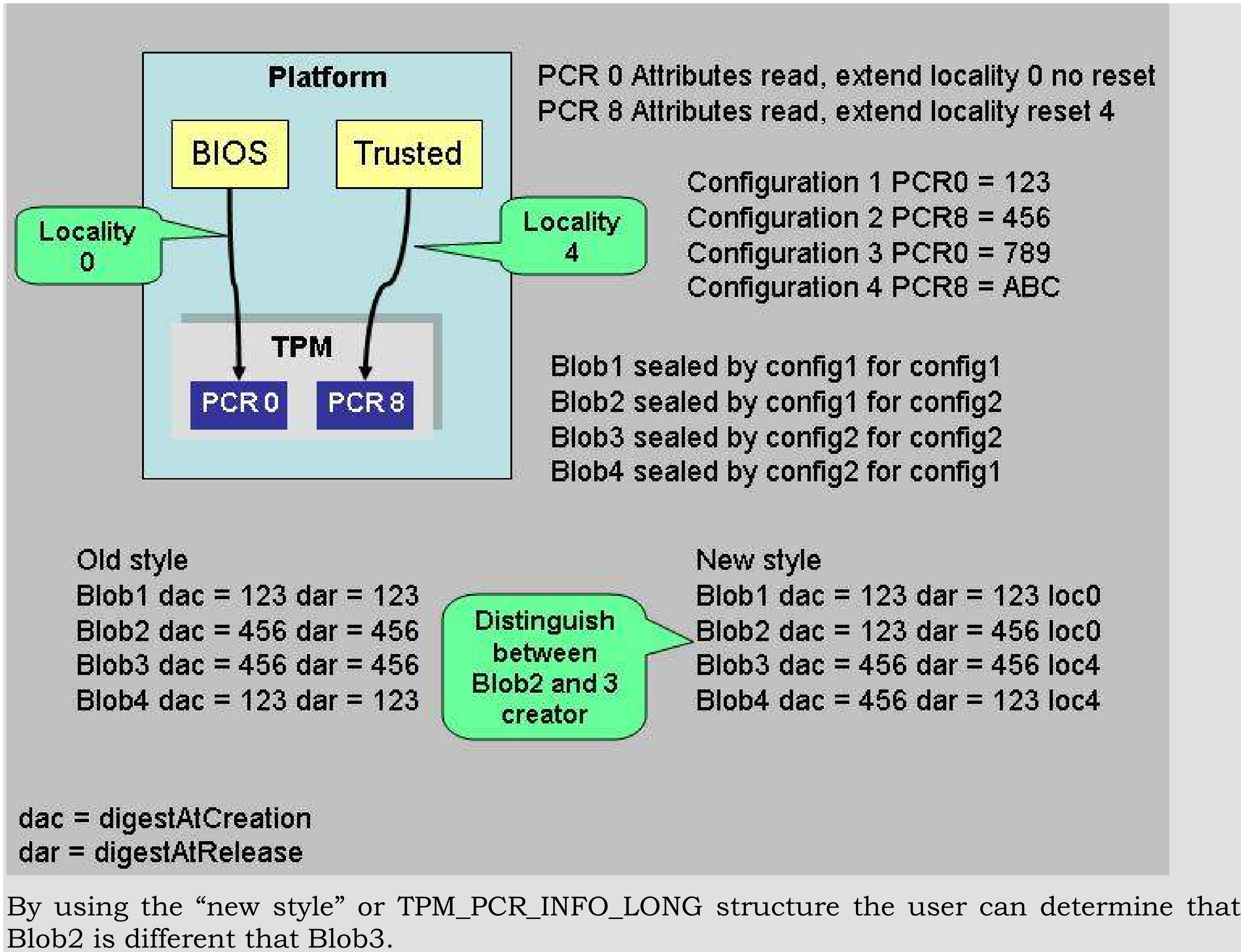
3401 **Definition of "Use"**

3402 See the definition of TPM_PCR_ATTRIBUTES for the attributes and the normative
3403 statements regarding the use of the attributes. The use of a configuration is when the TPM
3404 needs to ensure that the proper platform configuration is present. The first example is for
3405 Unseal, the TPM must only release the information sealed if the platform configuration
3406 matches the configuration specified by the seal creator. Here the use of locality is implicit in
3407 the PCR attributes, if PCR8 requires locality 2 to be present then the seal creator ensures
3408 that locality 2 is asserted by defining a configuration that uses PCR8.

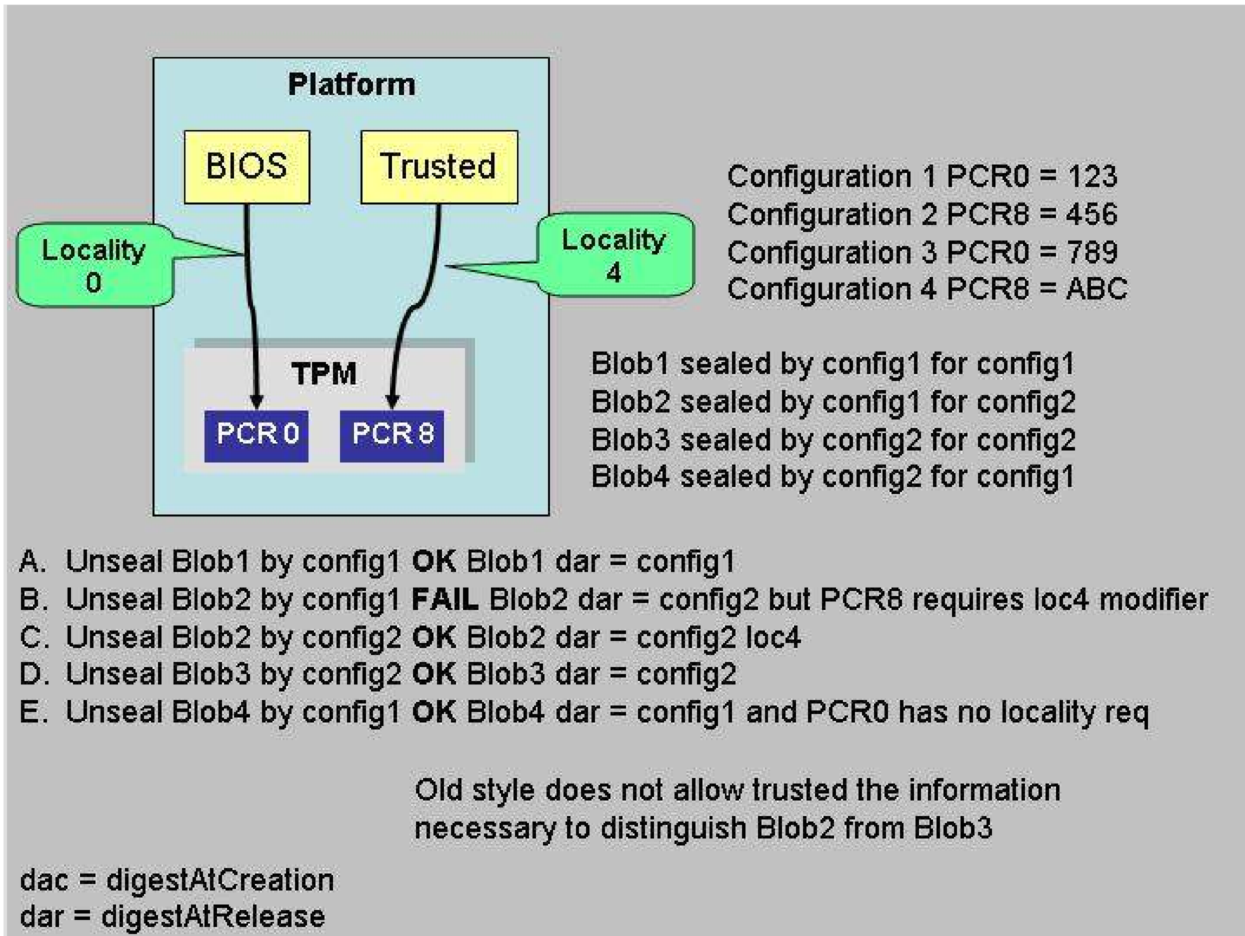
3409 The creation of a blob that specifies a configuration for use is not a "use" itself. So the SEAL
3410 command does is not a use for specifying the use of a PCR configuration.

3411

Copyright © TCG



By using the "new style" or TPM_PCR_INFO_LONG structure the user can determine that Blob2 is different that Blob3.



Case B is the only failure and this shows the use of the locality modifier and PCR locality attribute.

Additional attempts are obvious failures, config3 and config4 are unable to unseal any of the 4 blobs.

One example is illustrative of the problems of just specifying locality without an accompanying PCR. Assume Blob5 which specifies a dar of config1 and a locality 4 modifier. Now either config2 or config4 can unseal Blob5. In fact there is no way to restrict ANY process that gains access to locality 4 from performing the unseal. As many platforms will have no restrictions as to which process can load in locality 4 there is no additional benefit of specifying a locality modifier. If the sealer wants protections, they need to specify a PCR that requires a locality modifier.

Defining locality modifiers dynamically

This feature would enable the platform to specify how and when a locality modifier applies to a PCR. The current definition of PCR attributes has the values set in TPM manufacturing and static for all TPM in a specific platform type (like a PC).

Defining dynamic attributes would make the use of a PCR very difficult. The sealer would have to have some way of ensuring that their wishes were enforced and challengers would have to pay close attention to the current PCR attributes. For these reasons the setting of the PCR attributes is defined as a static operation made during the platform specific specification.

Copyright © TCG

3436 **End of informative comment**

3437 27.1 Validate Key for use

3438 **Start of informative comment**

3439 The following shows the order and checks done before the use of a key that has PCR or
3440 locality restrictions.

3441 Note that there is no check for the PCR registers on the DSAP session. This is due to the
3442 fact that DSAP checks for the continued validity of the PCR that are attached to the DSAP
3443 and any change causes the invalidation of the DSAP session.

3444 The checks must validate the locality of the DSAP session as the PCR registers in use could
3445 have locality restrictions.

3446 **End of informative comment**

3447 1. If the authorization session is DSAP

3448 a. If the DSAP -> localityAtRelease is not 0x1F (or in other words some localities are not
3449 allowed)

3450 i. Validate that TPM_STANY_FLAGS -> localityModifier is matched by DSAP ->
3451 pcrInfo -> localityAtRelease, on mismatch return TPM_BAD_LOCALITY

3452 b. If DSAP -> digestAtRelease is not 0

3453 i. Calculate the current digest and compare to digestAtRelease, return
3454 TPM_BAD_PCR on mismatch

3455 c. If the DSAP points to an ordinal delegation

3456 i. Check that the DSAP authorizes the use of the intended ordinal

3457 d. If the DSAP points to a key delegation

3458 i. Check that the DSAP authorizes the use of the key

3459 e. If the key delegated is a CMK key

3460 i. The TPM MUST check the CMK_DELEGATE restrictions

3461 2. Set LK to the loaded key that is being used

3462 3. If LK -> pcrInfoSize is not 0

3463 a. If LK -> pcrInfo -> releasePCRSelection identifies the use of one or more PCR

3464 i. Calculate H1 a TPM_COMPOSITE_HASH of the PCR selected by LK -> pcrInfo ->
3465 releasePCRSelection

3466 ii. Compare H1 to LK -> pcrInfo -> digestAtRelease on mismatch return
3467 TPM_WRONGPCRVAL

3468 b. If localityAtRelease is NOT 0x1F

3469 i. Validate that TPM_STANY_FLAGS -> localityModifier is matched by LK -> pcrInfo -
3470 > localityAtRelease on mismatch return TPM_BAD_LOCALITY

3471 4. Allow use of the key

3472 # 28. Non Volatile Storage

3473 **Start of informative comment**

3474 The TPM contains protected non-volatile storage. There are many uses of this type of area;
3475 however, a TPM needs to have a defined set of operations that touch any protected area.
3476 The idea behind these instructions is to provide an area that the manufacturers and owner
3477 can use for storing information in the TPM.

3478 The TCG will define a limited set of information that it sees a need of storing in the TPM.
3479 The TPM and platform manufacturer may add additional areas.

3480 The NV storage area has a limited use before it will no longer operate Hence the NV
3481 commands are under TPM Owner control.

3482 A defined set of indexes are available when no TPM Owner is present to allow TPM and
3483 platform manufacturers the ability to fill in values before a TPM Owner exists.

3484 To locate if an index is available, use TPM_GetCapability to return the index and the size of
3485 the area in use by the index.

3486 The area may not be larger than the TPM input buffer. The TPM will report the maximum
3487 size available to allocate.

3488 The storage area is an opaque area to the TPM. The TPM, other than providing the storage,
3489 does not review the internals of the area.

3490 To SEAL a blob, the creator of the area specifies the use of PCR registers to read the value.
3491 This is the exact property of SEAL.

3492 To obtain a signed indication of what is in a NV store area the caller would setup a
3493 transport session with logging on and then get the signed log. The log shows the parameters
3494 so the caller can validate that the TPM holds the value.

3495 There is an attribute, for each index, that defines the expected write scheme for the index.
3496 The TPM may handle data storage differently based on the write scheme attribute that
3497 defines the expected for the index. Whenever possible the NV memory should be allocated
3498 with the write scheme attribute set to update as one block and not as individual bytes.

3499 The non-volatile storage described here is defined by TPM_NV_DefineSpace. Other
3500 structures that a manufacturer might decide to store in non-volatile memory (e.g., PCRs,
3501 keys, the audit digest) are logically separate and do not affect the space available for the NV
3502 indexed storage described here.

3503 **End of informative comment**

3504 1. The TPM MUST support the NV commands. The TPM MUST support the NV area as
3505 defined by the TPM_NV_INDEX values.

3506 2. The TPM MAY manage the storage area using any allocation and garbage collection
3507 scheme.

3508 3. To remove an area from the NV store the TPM owner would use the
3509 TPM_NV_DefineSpace command with a size of 0. Any authorized user can change the
3510 value written in the NV store.

3511 4. The TPM MUST treat the NV area as a shielded location.

Copyright © TCG

3512 a. The TPM does not provide any additional protections (like additional encryption) to
3513 the NV area.

3514 5. If a write operation is interrupted, then the TPM makes no guarantees about the data
3515 stored at the specified index. It MAY be the previous value, MAY be the new value or
3516 MAY be undefined or unpredictable. After the interruption the TPM MAY indicate that
3517 the index contains unpredictable information.

3518 a. The TPM MUST ensure that in case of interruption of a write to an index that all
3519 other indexes are not affected

3520 6. Minimum size of NV area is platform specific. The maximum area is TPM vendor specific.

3521 7. A TPM MUST NOT use the NV area to store any data dependent on data structures
3522 defined in Part II of the TPM specifications, except for the NV Storage structures implied
3523 by required index values or reserved index values.

3524
28.1 NV storage design principles

3525 **Start of informative comment**

3526 This section lists the design principles that motivate the NV area in the TPM. There was the
3527 realization that the current design made use of NV storage but not necessarily efficiently.
3528 The DIR, BIT and other commands placed demands on the TPM designer and required
3529 areas that while allowing for flexible use reserved space most likely never used (like DIR for
3530 locality 1).

3531 The following are the design principles that drive the function definitions.

3532 1. Provide efficient use of NV area on the TPM. NV storage is a very limited resource and
3533 data stored in the NV area should be as small as possible.

3534 2. The TPM does not control, edit, validate or manipulate in any manner the information in
3535 the NV store. The TPM is merely a storage device. The TPM does enforce the access rules as
3536 set by the TPM Owner.

3537 3. Allocation of the NV area for a specific use must be under control of the TPM Owner.

3538 4. The TPM Owner, when defining the area to use, will set the access and use policy for the
3539 area. The TPM Owner can set AuthData values, delegations, PCR values and other controls
3540 on the access allowed to the area.

3541 5. There must be a capability to allow TPM and platform manufacturers to use this area
3542 without a TPM Owner being present. This allows the manufacturer to place information into
3543 the TPM without an onerous manufacturing flow. Information in this category would
3544 include EK credential and platform credential.

3545 6. The management and use of the NV area should not require a large number of ordinals.

3546 7. The management and use of the NV area should not introduce new operating strategies
3547 into the TPM and should be easy to implement.

3548 **End of informative comment**

3549
28.1.1 NV Storage use models

3550 **Start of informative comment**

3551 This informative section describes some of the anticipated use models and the attributes a
3552 user of the storage area would need to set.

3553

3554 **Owner authorized for all access**

3555 TPM_NV_DefineSpace: attributes = PER_OWERREAD || PER_OWNERWRITE

3556 WriteValue(TPM Owner Auth, data)

3557 ReadValue(TPM Owner Auth, data)

3558

3559 **Set AuthData value**

3560 TPM_NV_DefineSpace: attributes = PER_AUTHREAD || PER_AUTHWRITE, auth =
3561 authValue

3562 WriteValue(authValue, data)

3563 ReadValue(authValue, data)

3564

3565 **Write once, only way to change is to delete and redefine**

3566 TPM_NV_DefineSpace: attributes = PER_WRITEDEFINE

3567 WriteValue(size = x, data) // successful

3568 WriteValue(size = 0) // locks

3569 WriteValue(size = x) // fails

3570 …

3571 TPM_Startup(ST_Clear) // Does not affect lock

3572 WriteValue(size = x, data) // fails

3573

3574 **Write until specific index is locked, lock reset on Startup(ST_Clear)**

3575 TPM_NV_DefineSpace: index = 3, attributes = PER_WRITE_STCLEAR

3576 TPM_NV_DefineSpace: index = 5, attributes = PER_WRITE_STCLEAR

3577 WriteValue(index = 3, size = x, data) // successful

3578 WriteValue(index = 5, size = x, data) // successful

3579 WriteValue(index = 3, size = 0) // locks

3580 WriteValue(index = 3, size = x, data) // fails

3581 WriteValue(index = 5, size = x, data) // successful

3582 …

3583 TPM_Startup(ST_Clear) // clears lock

3584 WriteValue(index = 3, size = x, data) // successful

3585 WriteValue(index = 5, size = x, data) // successful

Copyright © TCG

Write until index 0 is locked, lock reset by Startup(ST_Clear)

TPM_NV_DefineSpace: attributes = PER_GLOBALLOCK, index = 5

TPM_NV_DefineSpace: attributes = PER_GLOBALLOCK, index = 3

WriteValue(index = 3, size = x, data) // successful

WriteValue(index = 5, size = x, data) // successful

WriteValue(index = 0) // sets SV -> bGlobalLock to TRUE

WriteValue(index = 3, size = x, data) // fails

WriteValue(index = 5, size = x, data) // fails

…

TPM_Startup(ST_Clear) // clears lock

WriteValue(index = 3, size = x, data) // successful

WriteValue(index = 5, size = x, data) // successful

End of informative comment

28.2 Use of NV storage during manufacturing

Start of informative comment

The TPM needs the ability to write values to the NV store during manufacturing. It is possible that the values written at this time would require authorization during normal TPM use. The actual enforcement of these authorizations during manufacturing would cause numerous problems for the manufacturer.

The TPM will not enforce the NV authorization restrictions until the execution of a TPM_NV_DefineSpace with the handle of TPM_NV_INDEX_LOCK.

The 'D' bit indicates an NV index defined (typically) during manufacturing and then locked. While nvLocked is FALSE, indices with the 'D' set can be defined, deleted, or redefined as desired. Once nvLocked is set TRUE, the 'D' bit indices are locked. They cannot be defined, deleted or redefined.

nvLocked has the lifetime of the endorsement key.

End of informative comment

1. The TPM MUST NOT enforce the NV authorizations (auth values, PCR etc.) prior to the execution of TPM_NV_DefineSpace with an index of TPM_NV_INDEX_LOCK
    a. While the TPM is not enforcing NV authorizations, the TPM SHALL allow the use of TPM_NV_DefineSpace in any operational state (disabled, deactivated)

29. Delegation Model

Start of informative comment

The TPM Owner is an entity with a single "super user" privilege to control TPM operation. Thus if any aspect of a TPM requires management, the TPM Owner must perform that task himself or reveal his privilege information to another entity. This other entity thereby obtains the privilege to operate all TPM controls, not just those intended by the Owner. Therefore the Owner often must have greater trust in the other entity than is strictly necessary to perform an arbitrary task.

This delegation model addresses this issue by allowing delegation of individual TPM Owner privileges (the right to use individual Owner authorized TPM commands) to individual entities, which may be trusted processes.

Basic requirements:

Consumer user does not need to enter or remember a TPM Owner password. This is an ease of use and security issue. Not remembering the password may lead to bad security practices, increased tech support calls and lost data.

Role based administration and separation of duty. It should be possible to delegate just enough Owner privileges to perform some administration task or carry out some duty, without delegating all Owner privileges.

TPM should support multiple trusted processes. When a platform has the ability to load and execute multiple trusted processes then the TPM should be able to participate in the protection of secrets and proper management of the processes and their secrets. In fact, the TPM most likely is the root of storage for these values. The TPM should enable the proper management, protection and distribution of values held for the various trusted processes that reside on the same platform.

Trusted processes may require restrictions. A fundamental security tenet is the principle of least privilege, that is, to limit process functionality to only the functions necessary to accomplish the task. This delegation model provides a building block that allows a system designer to create single purpose processes and then ensure that the process only has access to the functions that it requires to complete the task.

Maintain the current authorization structure and protocols. There is no desire to remove the current TPM Owner and the protocols that authorize and manage the TPM Owner. The capabilities are a delegation of TPM Owner responsibilities. The delegation allows the TPM Owner to delegate some or all of the actions that a TPM Owner can perform. The TPM Owner has complete control as to when and if the capability delegation is in use.

End of informative comment

29.1 Table Requirements

Start of informative comment

No ocean front property in table – We want the table to be virtually unlimited in size. While we need some storage, we do not want to pick just one number and have that be the min and max. This drives the need for the ability to save, off the TPM, delegation elements.

Revoking a delegation, does not affect other delegations – The TPM Owner may, at any time, determine that a delegation is no longer appropriate. The TPM Owner needs to be able

to ensure the revocation of all delegations in the same family. The TPM Owner also wants to ensure that revocation done in one family does not affect any other family of delegations.

Table seeded by OEM – The OEM should do the seeding of the table during manufacturing. This allows the OEM to ship the platform and make it easy for the platform owner to startup the first time. The definition of manufacturing in this context includes any time prior to or including the time the user first turns on the platform.

Table not tied to a TPM owner – The table is not tied to the existence of a TPM owner. This facilitates the seeding of the table by the OEM.

External delegations need authorization and assurance of revocation – When a delegation is held external to the TPM, the TPM must ensure authorization of the delegation when loading the delegation. Upon revocation of a family or other family changes the TPM must ensure that prior valid delegations are not successfully loaded.

90% case, no need for external store – The normal case should be that the platform does not need to worry about having external delegations. This drives the need for some NV storage to hold a minimum number of table rows.

End of informative comment

29.2 How this works

Start of informative comment

The existing TPM owner authorization model is that certain TPM commands require the authorization of the TPM Owner to operate. The authorization value is the TPM Owners token. Using the token to authorize the command is proof of TPM Ownership. There is only one token and knowledge of this token allows all operations that require proof of TPM Ownership.

This extension allows the TPM Owner to create a new AuthData value and to delegate some of the TPM Ownership rights to the new AuthData value.

The use model of the delegation is to create an authorization session (DSAP) using the delegated AuthData value instead of the TPM Owner token. This allows delegation to work without change to any current command.

The intent is to permit delegation of selected Owner privileges to selected entities, be they local or remote, separate from the current software environment or integrated into the current software environment. Thus Owner privileges may be delegated to entities on other platforms, to entities (trusted processes) that are part of the normal software environment on the Owner’s platform, or to a minimalist software environment on the Owner’s platform (created by booting from a CDROM, or special disk partition), for example.

Privileges may be delegated to a particular entity via definition of a particular process on the Owner’s platform (by dictating PCR values), and/or by stipulating a particular AuthData value. The resultant TPM_DELEGATE_OWNER_BLOB and any AuthData value must be passed by the Owner to the chosen entity.

Delegation to an external entity (not on the Owner’s platform) probably requires an AuthData value and a NULL PCR selection. (But the AuthData value might be sealed to a desired set of PCRs in that remote platform.)

3702 Delegation to a trusted process provided by the local OS requires a PCR that indicates the
3703 trusted process. The authorization token should be a fixed value (any well known value),
3704 since the OS has no means to safely store the authorization token without sealing that
3705 token to the PCR that indicates the trusted process. It is suggested that the value
3706 0x111…111 be used.

3707 Delegation to a specially booted entity requires either a PCR or an authorization token, and
3708 preferably both, to recognize both the process and the fact that the Owner wishes that
3709 process to execute.

3710 The central delegation data structure is a set of tables. These tables indicate the command
3711 ordinals delegated by the TPM Owner to a particular defined environment. The tables allow
3712 the distinction of delegations belonging to different environments.

3713 The TPM is capable of storing internally a few table elements to enable the passing of the
3714 delegation information from an entity that has no access to memory or storage of the
3715 defined environment.

3716 The number of delegations that the tables can hold is a dynamic number with the
3717 possibility of adding or deleting entries at any time. As the total number is dynamic, and
3718 possibly large, the TPM provides a mechanism to cache the delegations. The cache of a
3719 delegation must include integrity and confidentiality. The term for the encrypted cached
3720 entity is blob. The blob contains a counter (verificationCount) validated when the TPM loads
3721 the blob.

3722 An Owner uses the counter mechanism to prevent the use of undesirable blobs; they
3723 increment verificationCount inside the TPM and insert the current value of
3724 verificationCount into selected table elements, including temporarily loaded blobs. (This is
3725 the reason why a TPM must still load a blob that has an incorrect verificationCount.) An
3726 Owner can verify the delegation state of his platform (immediately after updating
3727 verificationCount) by keeping copies of the elements that have just been given the current
3728 value of verificationCount, signing those copies, and sending them to a third party.

3729 Verification probably requires interaction with a third party because acceptable table
3730 profiles will change with time and the most important reason for verification is suspicion of
3731 the state of a TOS in a platform. Such suspicion implies that the verification check must be
3732 done by a trusted security monitor (perhaps separate trusted software on another platform
3733 or separate trusted software on CDROM, for example). The signature sent to the third party
3734 must include a freshness value, to prevent replay attacks, and the security monitor must
3735 verify that a response from the third party includes that freshness value. In situations
3736 where the highest confidence is required, the third party could provide the response by an
3737 out-of-band mechanism, such as an automated telephone service with spoken confirmation
3738 of acceptability of platform state and freshness value.

3739 A challenger can verify an entire family using a single transport session with logging, that
3740 increments the verification count, updates the verification count in selected blobs, reads the
3741 tables and obtains a single transport session signature over all of the blobs in a family.

3742 If no Owner is installed, the delegation mechanisms are inoperative and third party
3743 verification of the tables is impossible, but tables can still be administered and corrected.
3744 (See later for more details.)

3745 To perform an operation using the delegation the entity establishes an authorization session
3746 and uses the delegated AuthData value for all HMAC calculations. The TPM validates the
3747 AuthData value, and in the case of defined environments checks the PCR values. If the

Copyright © TCG

3748 validation is successful, the TPM then validates that the delegation allows the intended
3749 operation.

3750 There can be at least two delegation rows stored in non-volatile storage inside a TPM, and
3751 these may be changed using Owner privilege or delegated Owner privilege. Each delegation
3752 table row is a member of a family, and there can be at least eight family rows stored in non-
3753 volatile storage inside a TPM. An entity belonging to one family can be delegated the
3754 privilege to create a new family and edit the rows in its own family, but no other family.

3755 In addition to tying together delegations, the family concept and the family table also
3756 provides the mechanism for validation and revocation of exported delegate table rows, as
3757 well as the mechanism for the platform user to perform validation of all delegations in a
3758 family.

3759 **End of informative comment**

3760 29.3 Family Table

3761 **Start of informative comment**

3762 The family table has three main purposes.

3763 1 - To provide for the grouping of rows in the TPM_DELEGATE_TABLE; entities identified in
3764 delegate table rows as belonging to the same family can edit information in the other
3765 delegate table rows with the same family ID. This allows a family to manage itself and
3766 provides an easier mechanism during upgrades.

3767 2 - To provide the validation and revocation mechanism for exported
3768 TPM_DELEGATE_ROWS and those stored on the TPM in the delegation table

3769 3 - To provide the ability to perform validation of all delegations in a family

3770 The family table must have eight rows, and may have more. The maximum number of rows
3771 is TPM vendor-defined and is available using the TPM_GetCapability command.

3772 As the family table has a limited number of rows, there is the possibility that this number
3773 could be insufficient. However, the ability to create a virtual amount of rows, like done for
3774 the TPM_DELEGATE_TABLE would create the need to have all of the validation and
3775 revocation mechanisms that the family table provides for the delegate table. This could
3776 become a recursive process, so for this version of the specification, the recursion stops at
3777 the family table.

3778 The family table contains four pieces of information: the family ID, the family label, the
3779 family verification count, and the family flags.

3780 The family ID is a 32-bit value that provides a sequence number of the families in use.

3781 The family label is a one-byte field that family table manager software would use to help
3782 identify the information associated with the family. Software must be able to map the
3783 numeric value associated with each family to the ASCII-string family name displayable in
3784 the user interface.

3785 The family verification count is a 32-bit sequence number that identifies the last outside
3786 verification and attestation of the family information.

3787 Initialization of the family table occurs by using the TPM_Delegate_Manage command with
3788 the TPM_FAMILY_CREATE option.

3789 The verificationCount parameter enables a TPM to check that all rows of a family in the
3790 delegate table are approved (by an external verification process), even if rows have been
3791 stored off-TPM.

3792 The family flags allow the use and administration of the family table row, and its associated
3793 delegate table rows.

3794 **Row contents**

3795 Family ID – 32-bits

3796 Row label – One byte

3797 Family verification count – 32-bits

3798 Family enable/disable use/admin flags – 32-bits

3799 **End of informative comment**

3800 29.4 Delegate Table

3801 **Start of informative comment**

3802 The delegate table has three main purposes, from the point of view of the TPM. This table
3803 holds:

3804 The list of ordinals allowable for use by the delegate

3805 The identity of a process that can use the ordinal list

3806 The AuthData value to use the ordinal list

3807 The delegate table has a minimum of two (2) rows; the maximum number of rows is TPM
3808 vendor-defined and is available using the TPM_GetCapability command. Each row
3809 represents a delegation and, optionally, an assignment of that delegation to an identified
3810 trusted process.

3811 The non-volatile delegate rows permit an entity to pass delegation rows to a software
3812 environment without regard to shared memory between the entity and the software
3813 environment. The size of the delegate table does not restrict the number of delegations
3814 because TPM_Delegate_CreateOwnerDelegation can create blobs for use in a DSAP session,
3815 bypassing the delegate table.

3816 The TPM Owner controls the tables that control the delegations, but (recursively) the TPM
3817 Owner can delegate the management of the tables to delegated entities. Entities belonging
3818 to a particular group (family) of delegation processes may edit delegate table entries that
3819 belong to that family.

3820 After creation of a delegation entry there is no restriction on the use of the delegation in a
3821 properly authorized session. The TPM Owner has properly authorized the creation of the
3822 delegation so the use of the delegation occurs whenever the delegate wishes to use it.

3823 The rows of the delegate table held in non-volatile storage are only changeable under TPM
3824 Owner authorization.

3825 The delegate table contains six pieces of information: PCR information, the AuthData value
3826 for the delegated capabilities, the delegation label, the family ID, the verification count, and
3827 a profile of the capabilities that are delegated to the trusted process identified by the PCR
3828 information.

Row Elements

ASCII label – Label that provides information regarding the row. This is not a sensitive item.

Family ID – The family that the delegation belongs to; this is not a sensitive item.

Verification count – Specifies the version, or generation, of this row; version validity information is in the family table. This is not a sensitive value.

Delegated capabilities – The capabilities granted, by the TPM Owner, to the identified process. This is not a sensitive item.

Authorization and Identity

The creator of the delegation sets the AuthData value and the PCR selection. The creator is responsible for the protection and dissemination of the AuthData value. This is a sensitive value.

End of informative comment

1. The TPM_DELEGATE_TABLE MUST have at least two (2) rows; the maximum number of table rows is TPM-vendor defined and MUST be reported in response to a TPM_GetCapability command
2. The AuthData value and the PCR selection must be set by the creator of the delegation

29.5 Delegation Administration Control

Start of informative comment

The delegate tables (both family and delegation) present some control problems. The tables must be initialized by the platform OEM, administered and controlled by the TPM Owner, and reset on changes of TPM Ownership. To provide this level of control there are three phases of administration with different functions available in the phases.

The three phases of table administration are; manufacturing (P1), no-owner (P2) and owner present (P3). These three phases allow different types of administration of the delegation tables.

Manufacturing (P1)

A more accurate definition of this phase is open, un-initialized and un-owned. It occurs after TPM manufacturing and as a result of TPM_OwnerClear or TPM_ForceClear.

In P1 TPM_Delegate_Manage can initialize and manage non-volatile family rows in the TPM. TPM_Delegate_LoadOwnerDelegation can load non-volatile delegation rows in the TPM.

Attacks that attempt to burnout the TPM's NV storage are frustrated by the NV store's own limits on the number of writes when no Owner is installed.

No-Owner (P2)

This phase occurs after the platform has been properly setup. The setup can occur in the platform manufacturing flow, during the first boot of the platform or at any time when the platform owner wants to lock the table settings down. There is no TPM Owner at this time.

TPM_Delegate_Manage locks both the family and delegation rows. This lock can be opened only by the Owner (after the Owner has been installed, obviously) or by the act of removing

3867 the Owner (even if no Owner is installed). Thus locked tables can be unlocked by asserting
3868 Physical Presence and executing TPM_ForceClear, without having to install an Owner.

3869 In P2, the relevant TPM_Delegate_xxx commands all return the error
3870 TPM_DELEGATE_LOCKED. This is not an issue as there is no TPM Owner to delegate
3871 commands, so the inability to change the tables or create delegations does not affect the
3872 use of the TPM.

3873 **Owned (P3)**

3874 In this phase, the TPM has a TPM Owner and the TPM Owner manages the table as the
3875 Owner sees fit. This phase continues until the removal of the TPM Owner.

3876 Moving from P2 to P3 is automatic upon establishment of a TPM Owner. Removal of the
3877 TPM Owner automatically moves back to P1.

3878 The TPM Owner always has the ability to administer any table. The TPM Owner may
3879 delegate the ability to manipulate a single family or all families. Such delegations are
3880 operative only if delegations are enabled.

3881 **End of informative comment**

3882 1. When DelegateAdminLock is TRUE the TPM MUST disallow any changes to the delegate
3883 tables

3884 2. With a TPM Owner installed, the TPM Owner MUST authorize all delegate table changes

3885 29.5.1 Control in Phase 1

3886 **Start of informative comment**

3887 The TPM starts life in P1. The TPM has no owner and the tables are empty. It is desirable
3888 for the OEM to initialize the tables to allow delegation to start immediately after the Owner
3889 decides to enable delegation. As the setup may require changes and validation, a simple
3890 mechanism of writing to the area once is not a valid option.

3891 TPM_Delegate_Manage and TPM_Delegate_LoadOwnerDelegation allow the OEM to fill the
3892 table, read the public parts of the table, perform reboots, reset the table and when finally
3893 satisfied as to the state of the platform, lock the table.

3894 Alternatively, the OEM can leave the tables NULL and turn off table administration leaving
3895 the TPM in an unloaded state waiting for the eventual TPM Owner to fill the tables, as they
3896 need.

3897 Flow to load tables

3898 Default values of DelegateAdminLock are set either during manufacturing or are the result
3899 of TPM_OwnerClear or TPM_ForceClear.

3900 TPM_Delegate_Manage verifies that DelegateAdminLock is FALSE and that there is no TPM
3901 Owner. The command will therefore load or manipulate the family tables as specified in the
3902 command.

3903 TPM_Delegate_LoadOwnerDelegation verifies that DelegateAdminLock is FALSE and no TPM
3904 owner is present. The command loads the delegate information specified in the command.

3905 **End of informative comment**

3906 ### 29.5.2 Control in Phase 2

3907 **Start of informative comment**

3908 In phase 2, no changes are possible to the delegate tables. The platform owner must install
3909 a TPM Owner and then manage the tables, or use TPM_ForceClear to revert to phase 1.

3910 **End of informative comment**

3911 ### 29.5.3 Control in Phase 3

3912 Start of informative comment

3913 The TPM_DELEGATE_TABLE requires commands that manage the table. These commands
3914 include filling the table, turning use of the table on or off, turning administration of the
3915 table on or off, and using the table.

3916 The commands are:

3917 **TPM_Delegate_Manage** – Manages the family table on a row-by-row basis: creates a new
3918 family, enables/disables use of a family table row and delegate table rows that share the
3919 same family ID, enables/disables administration of a family's rows in both the family table
3920 and the delegate table, and invalidates an existing family.

3921 **TPM_Delegate_CreateOwnerDelegation** increments the family verification count (if
3922 desired) and delegates the Owner's privilege to use a set of command ordinals, by creating a
3923 blob. Such blobs can be used as input data for TPM_DSAP or
3924 TPM_Delegate_LoadOwnerDelegation. Incrementing the verification count and creating a
3925 delegation must be an atomic operation. Otherwise no delegations are operative after
3926 incrementing the verification count.

3927 **TPM_Delegate_LoadOwnerDelegation** loads a delegate blob into a non-volatile delegate
3928 table row, inside the TPM.

3929 **TPM_Delegate_ReadTable** is used to read from the TPM the public contents of the family
3930 and delegate tables that are stored on the TPM.

3931 **TPM_Delegate_UpdateVerification** sets the verificationCount in an entity (a blob or a
3932 delegation row) to the current family value, in order that the delegations represented by that
3933 entity will continue to be accepted by the TPM.

3934 **TPM_Delegate_VerifyDelegation** loads a delegate blob into the TPM, and returns success
3935 or failure, depending on whether the blob is currently valid.

3936 **TPM_DSAP** – opens a deferred authorization session, using either an input blob (created by
3937 TPM_Delegate_CreateOwnerDelegation) or a cached blob (loaded by
3938 TPM_Delegate_LoadOwnerDelegation into one of the TPM's non-volatile delegation rows).

3939 **End of informative comment**

3940 ## 29.6 Family Verification

3941 **Start of informative comment**

3942 The platform user may wish to have confirmation that the delegations in use provide a
3943 coherent set of delegations. This process would require some evaluation of the processes
3944 granted delegations. To assist in this confirmation the TPM provides a mechanism to group

3945 all delegations of a family into a signed blob. The signed blob allows the verification agent to
3946 look at the delegations, the processes involved and make an assessment as the validity of
3947 the delegations. The third party then sends back to the platform owner the results of the
3948 assessment.

3949 To perform the creation of the signed blob the platform owner needs the ability to group all
3950 of the delegations of a single family into a transport session. The platform owner also wants
3951 an assurance that no management of the table is possible during the verification.

3952 This verification does not prove to a third party that the platform owner is not cheating.
3953 There is nothing to prevent the platform owner from performing the validation and then
3954 adding an additional delegation to the family.

3955 Here is one example protocol that retrieves the information necessary to validate the rows
3956 belonging to a particular family. Note that the local method of executing the protocol must
3957 prevent a man-in-the-middle attack using the nonce supplied by the user.

3958 The TPM Owner can increment the family verification count or use the current family
3959 verification count. Using the current family verification count carries the risk that
3960 unexamined delegation blobs permit undesirable delegations. Using an incremented
3961 verification count eliminates that risk. The entity gathering the verification data requires
3962 Owner authorization or access to a delegation that grants access to transport session
3963 commands, plus other commands depending on whether verificationCount is to be
3964 incremented. This delegation could be a trusted process that can use the delegations
3965 because of its PCR measurements, a remote entity that can use the delegations because the
3966 Owner has sent it a TPM_DELEGATE_OWNER_BLOB and AuthData value, or the host
3967 platform booted from a CDROM that can use the delegations because of its PCR
3968 measurements, and TPM_DELEGATE_OWNER_BLOB and AuthData value submitted by the
3969 Owner, for example.

3970 Verification using the current verificationCount

3971 The gathering entity requires access to a delegation that grants access to at least the
3972 ordinals to perform a transport session, plus TPM_Delegate_ReadTable and
3973 TPM_Delegate_VerifyDelegation.

3974 The TPM Owner creates a transport session with the "no other activity" attribute set. This
3975 ensures notification if other operations occur on the TPM during the validation process. (If
3976 other operations do occur, the validation processes may have been subverted.) All
3977 subsequent commands listed are performed using the transport session.

3978 TPM_Delegate_ReadTable displays all public values (including the permissions and PCR
3979 values) in the TPM.

3980 TPM_Delegate_VerifyDelegation loads each cached blob, with all public values (including the
3981 permissions and PCR values) in plain text.

3982 After verifying all blobs, TPM_ReleaseTransportSigned signs the list of transactions.

3983 The gathering entity sends the log of the transport session plus any supporting information
3984 to the validation entity, which evaluates the signed transport session log and informs the
3985 platform owner of the result of the evaluation. This could be an out-of-band process.

3986 Verification using an incremented verificationCount

3987 The gathering entity requires Owner authorization or access to a delegation that grants
3988 access to at least the ordinals to perform a transport session, plus

3989 TPM_Delegate_CreateOwnerDelegation, TPM_Delegate_ReadTable, and
3990 TPM_Delegate_UpdateVerification.

3991 The TPM Owner creates a transport session with the "no other activity" attribute set.

3992 To increment the count the TPM Owner (or a delegate) must use
3993 TPM_Delegate_CreateOwnerDelegation with increment == TRUE. That blob permits creation
3994 of new delegations or approval of existing tables and blobs. That delegation must set the
3995 PCRs of the desired (local) process and the desired AuthData value of the process. As noted
3996 previously, AuthData values should be a fixed value if the gathering entity is a trusted
3997 process that is part of the normal software environment.

3998 If new delegations are to be created, TPM_Delegate_CreateOwnerDelegation must be used
3999 with increment == FALSE.

4000 If existing blobs and delegation rows are to be reapproved,
4001 TPM_Delegate_UpdateVerification must be used to install the new value of verificationCount
4002 into those existing blobs and non-volatile rows. This exposes the blobs' public information
4003 (including the permissions and PCR values) in plain text to the transport session.

4004 TPM_Delegate_ReadTable then exposes all public values (including the permissions and
4005 PCR values) of tables to the transport session.

4006 Again, after verifying all blobs, TPM_ReleaseTransportSigned signs the list of transactions.

4007 **End of informative comment**

4008 29.7 Use of commands for different states of TPM

4009 **Start of informative comment**

4010 Use the ordinal table to determine when the various commands are available for use

4011 **End of informative comment**

4012 29.8 Delegation Authorization Values

4013 **Start of informative comment**

4014 This section describes why, when a PCR selection is set, the AuthData value may be a fixed
4015 value, and, when the PCR selection is null, the delegation creator must select an AuthData
4016 value.

4017 A PCR value is an indication of a particular (software) environment in the local platform.
4018 Either that PCR value indicates a trusted process or not. If the trusted process is to execute
4019 automatically, there is no point in allocating a meaningful AuthData value. (The only way
4020 the trusted process could store the AuthData value is to seal it to the process's PCR values,
4021 but the delegation mechanism is already checking the process's PCR values.) If execution of
4022 the trusted process is dependent upon the wishes of another entity (such as the Owner), the
4023 AuthData value should be a meaningful (private) value known only to the TPM, the Owner,
4024 and that other entity. Otherwise the AuthData value should be a fixed, well known, value.

4025 If the delegation is to be controlled from a remote platform, these simple delegation
4026 mechanisms provide no means for the platform to verify the PCRs of that remote platform,
4027 and hence access to the delegation must be based solely upon knowledge of the AuthData
4028 value.

End of informative comment

29.8.1 Using the authorization value

Start of informative comment

To use a delegation the TPM will enforce any PCR selection on use. The use definition is any command that uses the delegation authorization value to take the place of the TPM Owner authorization.

PCR Selection defined

In this case, the delegation has a PCR selection structure defined. Each time the TPM uses the delegation authorization value instead of the TPM Owner value the TPM would validate that the current PCR settings match the settings held in the delegation structure. The PCR selection includes the definition of localities and checks of locality occur with the checking of the PCR values. The TPM enforces use of the correct authorization value, which may or may not be a meaningful (private) value.

PCR selection NULL

In this case, the delegation has no PCR selection structure defined. The TPM does not enforce any particular environment before using the authorization value. Mere knowledge of the value is sufficient.

End of informative comment

29.9 DSAP description

Start of informative comment

The DSAP opens a deferred auth session, using either a TPM_DELEGATE_BLOB as input parameter or a reference to the TPM_DELEGATE_TABLE_ROW, stored inside the TPM. The DSAP command creates an ephemeral secret to authenticate a session. The purpose of this section is to illustrate the delegation of user keys or TPM Owner authorization by creating and using a DSAP session without regard to a specific command.

A key defined for a certain usage (e.g. TPM_KEY_IDENTITY) can be applied to different functions within the use model (e.g. TPM_Quote or TPM_CertifiyKey). If an entity knows the AuthData for the key (key.usageAuth) it can perform all the functions, allowed for that use model of that particular key. This entity is also defined as delegation creation entity, since it can initiate the delegation process. Assume that a restricted usage entity should only be allowed to execute a subset or a single functions denoted as TPM_Example, within the specific use model of a key. (e.g. Allow the usage of a TPM_IDENTITY_KEY only for Certifying Keys, but no other function). This use model points to the selection of the DSAP as the authorization protocol to execute the TPM_Example command.

To perform this scenario the delegation creation entity must know the AuthData for the key (key.usageAuth). It then has to initiate the delegation by creating a TPM_DELEGATE_KEY_BLOB via the TPM_Delegate_CreateKeyDelegation command. As a next step the delegation creation entity has to pass the TPM_DELEGATE_KEY_BLOB and the delegation AuthData (TPM_DELEGATE_SENSITIVE.authValue) to the restricted usage entity. The specification offers the TPM_DelTable_ReadAuth mechanism to perform this function. Other mechanisms may be used.

 Copyright © TCG

The restricted usage entity can now start an TPM_DSAP session by using the TPM_DELEGATE_KEY_BLOB as input.

For the TPM_Example command, the inAuth parameter provides the authorization to execute the command. The following table shows the commands executed, the parameters created and the wire formats of all of the information.

<inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne, inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode, ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order: authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

In addition to the two even nonces generated by the TPM (authLastNonceEven and nonceEven) that are used for TPM_OIAP, there is a third, labeled nonceEvenOSAP that is used to generate the shared secret. For every even nonce, there is also an odd nonce generated by the system.

4084

Caller	On the wire	Dir	TPM
Send TPM_DSAP	TPM_DSAP keyHandle nonceOddOSAP entityType entityValue	→	Decrypt sensitiveArea of entityValue If entityValue==TPM_ET_DEL_BLOB verify the integrity of the blob, and if a TPM_DELEGATE_KEY_BLOB is input verify that KeyHandle and entityValue match Create session & authHandle Generate authLastNonceEven Save authLastNonceEven with authHandle Generate nonceEvenOSAP Generate sharedSecret = HMAC(sensitiveArea.authValue., nonceEvenOSAP, nonceOddOSAP) Save keyHandle, sharedSecret with authHandle and permissions
Save authHandle, authLastNonceEven Generate sharedSecret = HMAC(sensitiveArea.authValue, nonceEvenOSAP, nonceOddOSAP) Save sharedSecret	authHandle, authLastNonceEven nonceEvenOSAP	←	Returns
Generate nonceOdd & save with authHandle. Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)			
Send TPM_Example	tag paramSize ordinal inArgOne inArgTwo authHandle nonceOdd continueAuthSession inAuth	→	Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Check if command ordinal of TPM_Example is allowed in permissions. If not return TPM_DISABLED_CMD Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

4085

4086

4087 Suppose now that the TPM user wishes to send another command using the same session
4088 to operate on the same key. For the purposes of this example, we will assume that the same
4089 ordinal is to be used (TPM_Example). To re-use the previous session, the
4090 continueAuthSession output boolean must be TRUE.

4091 The following table shows the command execution, the parameters created and the wire
4092 formats of all of the information.

4093 In this case, authLastNonceEven is the nonceEven value returned by the TPM with the
4094 output parameters from the first execution of TPM_Example.

Caller	On the wire	Dir	TPM
Generate nonceOdd Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Save nonceOdd with authHandle			
Send TPM_Example	tag paramSize ordinal inArgOne inArgTwo nonceOdd continueAuthSession inAuth	→	Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

4095

4096 The TPM user could then use the session for further authorization sessions or terminate it
4097 in the ways that have been described above in TPM_OIAP. Note that termination of the
4098 DSAP session causes the TPM to destroy the shared secret.

4099 **End of informative comment**

4100 1. The DSAP session MUST enforce any PCR selection on use. The use definition is any
4101 command that uses the delegation authorization value to take the place of the TPM
4102 Owner authorization.

30. Physical Presence

Start of informative comment

Physical presence is a signal from the platform to the TPM that indicates the operator manipulated the hardware of the platform. Manipulation would include depressing a switch, setting a jumper, depressing a key on the keyboard or some other such action.

TCG does not specify an implementation technique. The guideline is the physical presence technique should make it difficult or impossible for rogue software to assert the physical presence signal.

A PC-specific physical presence mechanism might be an electrical connection from a switch, or a program that loads during power on self-test.

End of informative comment

The TPM MUST support a signal from the platform for the assertion of physical presence. A TCG platform specific specification MAY specify what mechanisms assert the physical presence signal.

The platform manufacturer MUST provide for the physical presence assertion by some physical mechanism.

30.1 Use of Physical Presence

Start of informative comment

For control purposes there are numerous commands on the TPM that require TPM Owner authorization. Included in this group of commands are those that turn the TPM on or off and those that define the operating modes of the TPM. The TPM Owner always has complete control of the TPM. What happens in two conditions: there is no TPM Owner or the TPM Owner forgets the TPM Owner AuthData value. Physical presence allows for an authorization to change the state in these two conditions.

No TPM Owner

This state occurs when the TPM ships from manufacturing (it can occur at other times also). There is no TPM Owner. It is imperative to protect the TPM from remote software processes that would attempt to gain control of the TPM. To indicate to the TPM that the TPM operating state can change (allow for the creation of the TPM Owner) the human asserts physical presence. The physical presence assertion than indicates to the TPM that changing the operating state of the TPM is authorized.

Lost TPM Owner authorization

In the case of lost, or forgotten, authorization there is a TPM Owner but no way to manage the TPM. If the TPM will only operate with the TPM Owner authorization then the TPM is no longer controllable. Here the operator of the machine asserts physical presence and removes the current TPM Owner. The assumption is that the operator will then immediately take ownership of the TPM and insert a new TPM Owner AuthData value.

Operator disabling

Another use of physical presence is to indicate that the operator wants to disable the use of the TPM. This allows the operator to temporarily turn off the TPM but not change the permanent operating mode of the TPM as set by the TPM Owner.

Copyright © TCG

4144 **End of informative comment**

4145 # 31. TPM Internal Asymmetric Encryption

4146 **Start of Informative comment**

4147 For asymmetric encryption schemes, the TPM is not required to perform the blocking of
4148 information where that information cannot be encrypted in a single cryptographic
4149 operation. The schemes TPM_ES_RSAESOAEP_SHA1_MGF1 and TPM_ES_RSAESPKCSV15
4150 allow only single block encryption. When using these schemes, the caller to the TPM must
4151 perform any blocking and unblocking outside the TPM. It is the responsibility of the caller
4152 to ensure that multiple blocks are properly protected using a chaining mechanism.

4153 Note that there are inherent dangers associated with splitting information so that it can be
4154 encrypted in multiple blocks with an asymmetric key, and then chaining together these
4155 blocks together. For example, if an integrity check mechanism is not used, an attacker can
4156 encrypt his own data using the public key, and substitute this rogue block for one of the
4157 original blocks in the message, thus forcing the TPM to replace part of the message upon
4158 decryption.

4159 There is also a more subtle attack to discover the data encrypted in low-entropy blocks. The
4160 attacker makes a guess at the plaintext data, encrypts it, and substitutes the encrypted
4161 guess for the original block. When the TPM decrypts the complete message, a successful
4162 decryption will indicate that his guess was correct.

4163 There are a number of solutions which could be considered for this problem – One such
4164 solution for TPMs supporting symmetric encryption is specified in PKCS#7, section 10, and
4165 involves using the public key to encrypt a symmetric key, then using that symmetric key to
4166 encrypt the long message.

4167 For TPMs without symmetric encryption capabilities, an alternative solution may be to add
4168 random padding to each message block, thus increasing the block's entropy.

4169 **End of informative comment**

4170 1. For a TPM_UNBIND command where the parent key has pubKey.algorithmId equal to
4171 TPM_ALG_RSA and pubKey.encScheme set to TPM_ES_RSAESPKCSv15 the TPM SHALL
4172 NOT expect a PAYLOAD_TYPE structure to prepend the decrypted data.

4173 2. The TPM MUST perform the encryption or decryption in accordance with the
4174 specification of the encryption scheme, as described below.

4175 3. When a null terminated string is included in a calculation, the terminating null SHALL
4176 NOT be included in the calculation.

4177 ### 31.1.1 TPM_ES_RSAESOAEP_SHA1_MGF1

4178 1. The encryption and decryption MUST be performed using the scheme RSA_ES_OAEP
4179 defined in [PKCS #1v2.0: 7.1] using SHA1 as the hash algorithm for the encoding
4180 operation.

4181 2. Encryption

4182 a. The OAEP encoding P parameter MUST be the 4 character string "TCPA".

4183 b. While the TCG now controls this specification the string value will NOT change to
4184 allow for interoperability and backward compatibility with TCPA 1.1 TPM's

Copyright © TCG

c. If there is an error with the encryption, the TPM must return the error TPM_ENCRYPT_ERROR.

3. Decryption

a. The OAEP decoding P parameter MUST be the 4 character string "TCPA".

b. While the TCG now controls this specification the string value will NOT change to allow for interoperability and backward compatibility with TCPA 1.1 TPM's

c. If there is an error with the decryption, the TPM must return the error TPM_DECRYPT_ERROR.

31.1.2 TPM_ES_RSAESPKCSV15

1. The encryption MUST be performed using the scheme RSA_ES_PKCSV15 defined in [PKCS #1v2.0: 7.2].
2. Encryption
   a. If there is an error with the encryption, return the error TPM_ENCRYPT_ERROR.
3. Decryption
   a. If there is an error with the decryption, return the error TPM_DECRYPT_ERROR.

31.1.3 TPM_ES_SYM_CTR

Start of informative comment

This defines an encryption mode in use with symmetric algorithms. The actual definition is at

http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

The underlying symmetric algorithm may be AES128, AES192, or AES256. The definition for these algorithms is in the NIST document Appendix E.

The method of incrementing the counter value is different from that used by some standard crypto libraries (e.g. openSSL, Java JCE) that increment the entire counter value. TPM users should be aware of this to avoid errors when the counter wraps.

End of informative comment

1. Given a current counter value, the next counter value is obtained by treating the lower 32 bits of the current counter value as an unsigned 32-bit integer x, then replacing the lower 32 bits of the current counter value with the bits of the incremented integer (x + 1) mod 2^32. This method is described in Appendix B.1 of the NIST document (b=32).

31.1.4 TPM_ES_SYM_OFB

Start of informative comment

This defines an encryption mode in use with symmetric algorithms. The actual definition is at

http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

4220 The underlying symmetric algorithm may be AES128, AES192, or AES256. The definition
4221 for these algorithms is in the NIST document Appendix E.

4222 **End of informative comment**

4223 31.2 TPM Internal Digital Signatures

4224 **Start of informative comment**

4225 These values indicate the approved schemes in use by the TPM to generate digital
4226 signatures.

4227 TPM 1.1 included only _SHA1 keys. These allowed the TPM_Sign command to sign a hash
4228 with no structure. This signature scheme is retained for backward compatibility.

4229 TPM 1.2 added _INFO keys to ensure that a structure, rather than a plain hash, is always
4230 signed. For TPM_Sign, this signature scheme signs a new TPM_SIGN_INFO structure.
4231 Other ordinals, such as (e.g., TPM_GetAuditDigestSigned, TPM_CertifyKey, TPM_Quote, etc.)
4232 inherently sign a structure, so the _SHA1 and _INFO signature schemes produce an
4233 identical result.

4234 **End of informative comment**

4235 The TPM MUST perform the signature or verification in accordance with the specification of
4236 the signature scheme, as described below.

4237 31.2.1 TPM_SS_RSASSAPKCS1v15_SHA1

4238 **Start of informative comment**

4239 This signature scheme prepends an OID to a SHA-1 digest. The OID, as specified in the
4240 normative, is as follows:

4241 PKCS#1 v2.0: 8.1 says to encode the message per PKCS#1 v2.0: 9.2.1.

4242 PKCS#1 v2.0: 9.2.1 says to apply the digest and then add the algorithm ID per Section 11.

4243 PKCS#1 v2.0: Section 11.2.3 for SHA-1 says

4244 {iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) 26 }

4245 and also

4246 For each OID, the parameters field associated with this OID in an AlgorithmIdentifier
4247 shall have type NULL.

4248 The DER/BER Guide says that the first sub-identifiers are coded as 40 * value1 + value2.

4249 Thus, the OID becomes (with comments):

```
4250 0x30 SEQUENCE
4251 0x21 33 bytes
4252  0x30 SEQUENCE
4253  0x09 9 bytes
4254   0x06 OID
4255   0x05 5 bytes
4256    0x2b 43 = 40 * 1 (iso) + 3 (identified-organization)
4257    0x0e 14 from 11.2.3
4258    0x03  3 from 11.2.3
```

Copyright © TCG

4259 0x02 2 from 11.2.3
4260 0x1a 26 from 11.2.3
4261 0x05 NULL (parameters)
4262 0x00 0 bytes
4263 0x04 OCTET
4264 0x14 20 bytes (the SHA-1 digest to follow)

4265 **End of informative comment**

4266 1. The signature MUST be performed using the scheme RSASSA-PKCS1-v1.5 defined in
4267 [PKCS #1v2.0: 8.1] using SHA1 as the hash algorithm for the encoding operation.

4268 31.2.2 TPM_SS_RSASSAPKCS1v15_DER

4269 **Start of informative comment**

4270 This signature scheme is designed to permit inclusion of DER coded information before
4271 signing, which is inappropriate for most TPM capabilities

4272 **End of informative comment**

4273 1. The signature MUST be performed using the scheme RSASSA-PKCS1-v1.5 defined in
4274 [PKCS #1v2.0: 8.1]. The caller must properly format the area to sign using the DER
4275 rules. The provided area maximum size is k-11 octets.

4276 2. TPM_Sign SHALL be the only TPM capability that is permitted to use this signature
4277 scheme. If a capability other than TPM_Sign is requested to use this signature scheme,
4278 it SHALL fail with the error code TPM_INAPPROPRIATE_SIG

4279 31.2.3 TPM_SS_RSASSAPKCS1v15_INFO

4280 **Start of informative comment**

4281 This signature scheme is designed to permit signatures on arbitrary information but also
4282 protect the signature mechanism from being misused.

4283 **End of informative comment**

4284 1. The scheme MUST work just as TPM_SS_RSASSAPKCS1v15_SHA1 except in the
4285 TPM_Sign command

4286 a. In the TPM_Sign command the scheme MUST use a properly constructed
4287 TPM_SIGN_INFO structure, and hash it before signing

4288 31.2.4 Use of Signature Schemes

4289 **Start of informative comment**

4290 The TPM_SS_RSASSAPKCS1v15_INFO scheme is a new addition for 1.2. It causes a new
4291 functioning for 1.1 and 1.2 keys. The following details the use of the new scheme and how
4292 the TPM handles signatures and hashing

4293 **End of informative comment**

4294 1. For the commands (TPM_GetAuditDigestSigned, TPM_TickStampBlob,
4295 TPM_ReleaseTransportSigned):

4296 a. The TPM MUST create a TPM_SIGN_INFO and sign using the
4297 TPM_SS_RSASSAPKCS1v15_SHA1 scheme for either _SHA1 or _INFO keys.

4298 2. For the commands (TPM_CMK_CreateTicket, TPM_CertifyKey, TPM_CertifyKey2,
4299 TPM_MakeIdentity, TPM_Quote, TPM_Quote2):

4300 a. Create the structure as defined by the command and sign using the
4301 TPM_SS_RSASSAPKCS1v15_SHA1 scheme for either _SHA1 or _INFO keys.

4302 3. For TPM_Sign:

4303 a. Create the structure as defined by the command and key scheme

4304 b. If key->sigScheme is TPM_SS_RSASSAPKCS1v15_SHA1, sign the 20 byte parameter

4305 c. If key->sigScheme is TPM_SS_RSASSAPKCS1v15_DER, sign the DER value.

4306 d. If key->sigScheme is TPM_SS_RSASSAPKCS1v15_INFO, sign any value using the
4307 TPM_SIGN_INFO structure.

4308 4. When data is signed and the data comes from INSIDE the TPM, the TPM MUST do the
4309 hash, and prepend the DER encoding correctly before performing the padding and
4310 private key operation.

4311 5. When data is signed and the data comes from OUTSIDE the TPM, the software, not the
4312 TPM, MUST do the hash.

4313 6. When the TPM knows, or is told by implication, that the hash used is SHA-1, the TPM
4314 MUST prepend the DER encoding correctly before performing the padding and private
4315 key operation

4316 7. When the TPM does not know, or told by implication, that the hash used is SHA-1, the
4317 software, not the TPM) MUST provide the DER encoding to be prepended.

4318 8. The TPM MUST perform the padding and private key operation in any signing operations
4319 it does.

 Copyright © TCG

32. Key Usage Table

Start of informative comment

Asymmetric keys (e.g., RSA keys) can do two basic functions: sign/verify and encrypt/decrypt.

TPM_KEY_SIGNING and TPM_KEY_IDENTITY do signature functions.

TPM_KEY_STORAGE, TPM_KEY_BIND, TPM_KEY_MIGRATE, and TPM_KEY_AUTHCHANGE do encryption functions.

End of informative comment

This table summarizes the types of keys associated with a given TPM command.

It is the responsibility of each command to check the key usage prior to executing the command

			First Key						Second Key					
Name	First Key	Second Key	SIGNING	STORAGE	IDENTITY	AUTHCHG	BIND	LEGACY	SIGNING	STORAGE	IDENTITY	AUTHCHG	BIND	LEGACY
TPM_ActivateIdentity	idKey				x									
TPM_CertifyKey	certKey	inKey	x		x			x	x	x	x		x	x
TPM_CertifyKey2 (Note 3)	inKey	certKey	x	x	x		x	x	x		x			x
TPM_CertifySelfTest	key		x		x			x						
TPM_ChangeAuth	parent	blob		x					2	2	2	2	2	2
TPM_ChangeAuthAsymFinish	parent	ephemeral		x								x		
TPM_ChangeAuthAsymStart	idKey	ephemeral			x							x		
TPM_CMK_ConvertMigration	parent			x										
TPM_CMK_CreateBlob	parent			x										
TPM_CMK_CreateKey	parent			x										
TPM_ConvertMigrationBlob	parent			x										
TPM_CreateMigrationBlob	parent	blob		x					2	2	2	2	2	2
TPM_CreateWrapKey	parent			x										
TPM_Delegate_CreateKeyDelegation	key		x	x	x	x	x	x						
TPM_DSAP	entity		x	x	x	x	x	x						
TPM_EstablishTransport	key			x				x						
TPM_GetAuditDigestSigned	certKey		x		x			x						
TPM_GetAuditEventSigned	certKey		x					x						
TPM_GetCapabilitySigned	key		x		x			x						

TPM_GetPubKey	key		x	x	x	x	x	x						
TPM_KeyControlOwner	key		x	x	x		x	x						
TPM_LoadKey2	parent	inKey		x					x	x	x		x	x
TPM_LoadKey	parent	inKey		x					x	x	x		x	x
TPM_MigrateKey	maKey			1										
TPM_OSAP	entity		x	x	x	x	x	x						
TPM_Quote	key		x		x			x						
TPM_Quote2	key		x		x			x						
TPM_Seal	key			x										
TPM_Sealx	key			x										
TPM_Sign	key		x					x						
TPM_UnBind	key						x	x						
TPM_Unseal	parent			x										
TPM_ReleaseTransportSigned	key		x											
TPM_TickStampBlob	key		x		x			x						

4331 **Notes**

4332 1 – Key is not a storage key but TPM_MIGRATE_KEY

4333 2 – TPM unable to determine key type

4334 3 – The order is correct; the reason is to support a single auth version.

4335 # 33. Direct Anonymous Attestation

4336 **Start of informative comment**

4337 TPM_DAA_Join and TPM_DAA_Sign are highly resource intensive commands. They require
4338 most of the internal TPM resources to accomplish the complete set of operations. A TPM
4339 may specify that no other commands are possible during the join or sign operations. To
4340 allow other operations to occur, the TPM does allow the TPM_SaveContext command to save
4341 off the current join or sign operation.

4342 Operations that occur during a join or sign result in the loss of the join or sign session in
4343 favor of the interrupting command.

4344 **End of informative comment**

4345 1. The TPM MUST support one concurrent TPM_DAA_Join or TPM_DAA_Sign session. The
4346 TPM MAY support additional sessions

4347 2. The TPM MAY invalidate a join or sign session upon the receipt of any additional
4348 command other than the join/sign or TPM_SaveContext

4349 ## 33.1 TPM_DAA_JOIN

4350 **Start of informative comment**

4351 TPM_DAA_Join creates new JOIN data. If a TPM supports only one JOIN/SIGN operation,
4352 TPM_DAA_Join invalidates any previous DAA attestation information inside a TPM. The
4353 JOIN phase of a DAA context requires a TPM to communicate with an issuer.
4354 TPM_DAA_Join outputs data to be sent to an issuing authority and receives data from that
4355 issuing authority. The operation potentially requires several seconds to complete, but is
4356 done in a series of atomic stages and TPM_SaveContext/TPM_LoadContext can be used to
4357 cache data off-TPM in between atomic stages.

4358 The JOIN process is designed so a TPM will normally receive exactly the same DAA
4359 credentials from a given issuer, no matter how many times the JOIN process is executed
4360 and no matter whether the issuer changes his keys. This property is necessary because an
4361 issuer must give DAA credentials to a platform after verifying that the platform has the
4362 architecture of a trusted platform. Unless the issuer repeats the verification process, there
4363 is no justification for giving different DAA credentials to the same platform. Even after
4364 repeating the verification process, the issuer should give replacement (different) DAA
4365 credentials only when it is necessary to retire the old DAA credentials. Replacement DAA
4366 credentials erase the previous DAA history of the platform, at least as far as the DAA
4367 credentials from that issuer are concerned. Replacement might be desirable, as when a
4368 platform changes hands, for example, in order to eliminate any association via DAA between
4369 the seller and the buyer. On the other hand, replacement might be undesirable, since it
4370 enables a rogue to rejoin a community from which he has been barred. Replacement is done
4371 by submitting a different "count" value to the TPM during a JOIN process. A platform may
4372 use any value of "count" at any time, in any order, but only "counts" accepted by the issuer
4373 will elicit DAA credentials from that issuer.

4374 The TPM is forced to verify an issuer's public parameters before using an issuer's public
4375 parameters. This verification provides proof that the public parameters (which include a
4376 public key) were approved by an entity that knows the private key corresponding to that
4377 public key; in other words that the JOIN has previously been approved by the issuer. This

4378 verification is necessary to prevent an attack by a rogue using a genuine issuer's public
4379 parameters, which could reveal the secret created by the TPM using those public
4380 parameters. Verification uses a signature (provided by the issuer) over the public
4381 parameters.

4382 The exponent of the issuer's key is fixed at 2^16+1, because this is the only size of exponent
4383 that a TPM is required to support. The modulus of the issuer's public key is used to create
4384 the pseudonym with which the TPM contacts the issuer. Hence, the TPM cannot produce
4385 the same pseudonym for different issuers (who have different keys). The pseudonym is
4386 always created using the issuer's first key, even if the issuer changes keys, in order to
4387 produce the property described earlier. The issuer proves to the TPM that he has the right
4388 to use that first key to create a pseudonym by creating a chain of signatures from the first
4389 key to the current key, and submitting those signatures to the TPM. The method has the
4390 desirable property that only signatures and the most recent private key need be retained by
4391 the issuer: once the latest link in the signature chain has been created, previous private
4392 keys can be discarded.

4393 The use of atomic operations minimizes the contiguous time that a TPM is busy with
4394 TPM_DAA_Join and hence unavailable for other commands. JOIN can therefore be done as
4395 a background activity without inconveniencing a user. The use of atomic operations also
4396 minimizes the peak value of TPM resources consumed by the JOIN phase.

4397 The use of atomic operations introduces a need for consistency checks, to ensure that the
4398 same parameters are used in all atomic operations of the same JOIN process.
4399 DAA_tpmSpecific therefore contains a digest of the associated DAA_issuerSettings
4400 structure, and DAA_session contains a digest of associated DAA_tpmSpecific and
4401 DAA_joinSession structures. Each atomic operation verifies digests to ensure use of
4402 mutually consistent sets of DAA_issuerSettings, DAA_tpmSpecific, DAA_session, and
4403 DAA_joinSession data.

4404 JOIN operations and data structures are designed to minimize the amount of data that
4405 must be stored on a TPM in between atomic operations, while ensuring use of mutually
4406 consistent sets of data. Digests of public data are held in the TPM between atomic
4407 operations, instead of the actual public data (if a digest is smaller than the actual data). In
4408 each atomic operation, consistency checks verify that any public data loaded and used in
4409 that operation matches the stored digest. Thus non-secret DAA_generic_X parameters
4410 (loaded into the TPM only when required), are checked using digests DAA_digest_X
4411 (preloaded into the TPM in the structure DAA_issuerSettings).

4412 JOIN includes a challenge from the issuer, in order to defeat simple Denial of Service
4413 attacks on the issuer's server by rogues pretending to be arbitrary TPMs.

4414 A first group of atomic operations generate all TPM-data that must be sent to the issuer.
4415 The platform performs other operations (that do not need to be trusted) using the TPM-data,
4416 and sends the resultant data to the issuer. The issuer sends values u2 and u3 back to the
4417 TPM. A second group of atomic operations accepts this data from the issuer and completes
4418 the protocol.

4419 The TPM outputs encrypted forms of DAA_tpmSpecific, v0 and v1. These encrypted data are
4420 later interpreted by the same TPM and not by any other entity, so any manufacturer-
4421 specific wrapping can be used. It is suggested, however, that enc(DAA_tpmSpecific) or
4422 enc(v0) or enc(v1) data should be created by adapting a TPM_CONTEXT_BLOB structure.

4423 After executing TPM_DAA_Join, it is prudent to perform TPM_DAA_Sign, to verify that the
4424 JOIN process completed correctly. A host platform may choose to verify JOIN by performing
4425 TPM_DAA_Sign as both the target and the verifier (or could, of course, use an external
4426 verifier).

4427 **End of informative comment**

4428 33.2 TPM_DAA_Sign

4429 **Start of informative comment**

4430 TPM_DAA_Sign responds to a challenge and proves the attestation held by a TPM without
4431 revealing the attestation held by that TPM. The operation is done in a series of atomic
4432 stages to minimize the contiguous time that a TPM is busy and hence unavailable for other
4433 commands. TPM_SaveContext can be used to save a DAA context in between atomic stages.
4434 This enables the response to the challenge to be done as a background activity without
4435 inconveniencing a user, and also minimizes the peak value of TPM resources consumed by
4436 the process.

4437 The use of atomic operations introduces a need for consistency checks, to ensure that the
4438 same parameters are used in all atomic operations of the same SIGN process.
4439 DAA_tpmSpecific therefore contains a digest of the associated DAA_issuerSettings
4440 structure, and DAA_session contains a digest of associated DAA_tpmSpecific structure.
4441 Each atomic operation verifies these digests and hence ensures use of mutually consistent
4442 sets of DAA_issuerSettings, DAA_tpmSpecific, and DAA_session data.

4443 SIGN operations and data structures are designed to minimise the amount of data that
4444 must be stored on a TPM in between atomic operations, while ensuring use of mutually
4445 consistent sets of data. Digests of public and private data are held in the TPM between
4446 atomic operations, instead of the actual public or private data (if a digest is smaller than the
4447 actual data). At each atomic operation, consistency checks verify that any data loaded and
4448 used in that operation matches the stored digest. Thus parameters DAA_digest_X are
4449 digests (preloaded into the TPM in the structure DAA_issuerSettings) of non-secret
4450 DAA_generic_X parameters (loaded into the TPM only when required), for example.

4451 The design enables the use of any number of issuer DAA-data, private DAA-data, and so on.
4452 Strictly, the design is that the *TPM* puts no limit on the number of sets of issuer DAA-data
4453 or sets of private DAA-data, or restricts what set is in the TPM at any time, but supports
4454 only one DAA-context in the TPM at any instant. Any number of DAA-contexts can, of
4455 course, be swapped in and out of the TPM using TPM_SaveContext/TPM_LoadContext, so
4456 applications do not perceive a limit on the number of DAA contexts.

4457 TPM_DAA_Sign accepts a freshness challenge from the verifier and generates all TPM-data
4458 that must be sent to the verifier. The platform performs other operations (that do not need
4459 to be trusted) using the TPM-data, and sends the resultant data to the verifier. At one stage,
4460 the TPM incorporates a loaded public (non-migratable) key into the protocol. This is
4461 intended to permit the setup of a session, for any specific purpose, including doing the
4462 same job in TPM_ActivateIdentity as the EK.

4463 **End of informative comment**

4464 33.3 DAA Command summary

4465 **Start of informative comment**

4466 The following is a conceptual summary of the operations that are necessary to setup a TPM
4467 for DAA, execute the JOIN process, and execute the SIGN process.

4468 The summary is partitioned according to the "stages" of the actual TPM commands. Thus,
4469 the operations listed in JOIN under stage-2 briefly describe the operation of TPM_DAA_Join
4470 at stage-2, for example.

4471 This summary is in place to help in the connection between the mathematical definition of
4472 DAA and this implementation in a TPM.

4473 **End of informative comment**

4474 33.3.1 TPM setup

4475 1. A TPM generates a TPM-specific secret S (160-bit) from the RNG and stores S in
4476 nonvolatile store on the TPM. This value will never be disclosed and changed by the
4477 TPM.

4478 33.3.2 JOIN

4479 **Start of informative comment**

4480 This entire section is informative

4481 1. When the following is performed, this process does not increment the stage counter.

4482 a. TPM imports a non-secret values n0 (2048-bit).

4483 b. TPM computes a non-secret value N0 (160-bit) = H(n0).

4484 c. TPM computes a TPM-specific secret DAA_rekey (160-bit) = H(S, H(n0)).

4485 d. TPM stores a self-consistent set of (N0, DAA_rekey)

4486 2. The following is performed 0 or several times: (Note: If the stage mechanism is being
4487 used, then this branch does not increment the stage counter.)

4488 a. TPM imports

4489 i. a self consistent set of (N0, DAA_rekey)

4490 ii. a non-secret value DAA_SEED_KEY (2048-bit)

4491 iii. a non-secret value DEPENDENT_SEED_KEY (2048-bit)

4492 iv. a non-secret value SIG_DSK (2048-bit)

4493 b. TPM computes DIGEST (160-bit) = H(DAA_SEED_KEY)

4494 c. If DIGEST != N0, TPM refuses to continue

4495 d. If DIGEST == N0, TPM verifies validity of signature SIG_DSK on
4496 DEPENDENT_SEED_KEY with key (DAA_SEED_KEY, e0 (= 2^16 + 1)) by using
4497 TPM_Sign_Verify (based on PKCS#1 2.0). If check fails, TPM refuses to continue.

4498 e. TPM sets N0 = H(DEPENDENT_SEED_KEY)

4499 f. TPM stores a self consistent set of (N0, DAA_JOIN)

4500 3. Stage 2

4501 a. TPM imports a set of values, including

Copyright © TCG

4502 i. a non-secret value n0 (2048-bit),

4503 ii. a non-secret value R0 (2048-bit),

4504 iii. a non-secret value R1 (2048-bit),

4505 iv. a non-secret value S0 (2048-bit),

4506 v. a non-secret value S1 (2048-bit),

4507 vi. a non-secret value n (2048-bit),

4508 vii. a non-secret value n1 (1024-bit),

4509 viii. a non-secret value gamma (2048-bit),

4510 ix. a non-secret value q (208-bit),

4511 x. a non-secret value COUNT (8-bit),

4512 xi. a self consistent set of (N0, DAA_rekey).

4513 xii. TPM saves them as part of a new set A.

4514 b. TPM computes DIGEST (160-bit) = H(n0)

4515 c. If DIGEST != N0, TPM refuses to continue.

4516 d. If DIGEST == N0, TPM computes DIGEST (160-bit) = H(R0, R1, S0, S1, n, n1, Γ, q)

4517 e. TPM imports a non-secret value SIG_ISSUER_KEY (2048-bit).

4518 f. TPM verifies validity of signature SIG_ISSUER_KEY (2048-bit) on DIGEST with key (n0,
4519 e0) by using TPM_Sign_Verify (based on PKCS#1 2.0). If check fails, TPM refuses to
4520 continue.

4521 g. TPM computes a TPM-specific secret f (208-bit) = H(DAA_rekey, COUNT,
4522 0)||H(DAA_rekey, COUNT, 1) mod q.

4523 h. TPM computes a TPM-specific secret f0 (104-bit) = f mod 2104.

4524 i. TPM computes a TPM-specific secret f1 (104-bit) = f >> 104.

4525 j. TPM save f, f0 and f1 as part of set A.

4526 4. Stage 3

4527 a. TPM generates a TPM-specific secret u0 (1024-bit) from the RNG.

4528 b. TPM generates a TPM-specific secret u'1 (1104-bit) from the RNG.

4529 c. TPM computes u1 (1024-bit) = u'1 mod n1.

4530 d. TPM stores u0 and u1 as part of set A.

4531 5. Stage 4

4532 a. TPM computes a non-secret value P1 (2048-bit) = (R0^f0) mod n and stores P1 as part of
4533 set A.

4534 6. Stage 5

4535 a. TPM computes a non-secret value P2 (2048-bit) = P1*(R1^f1) mod n, stores P2 as part of
4536 set A and erases P1 from set A.

4537 7. Stage 6

4538 a. TPM computes a non-secret value P3 (2048-bit) = P2*(S0^u0) mod n, stores P3 as part of
4539 set A and erases P2 from set A.

4540 8. Stage 7

4541 a. TPM computes a non-secret value U (2048-bit) = P3*(S1^u1) mod n.

4542 b. TPM erases P3 from set A

4543 c. TPM computes and saves U1 (160-bit) = H(U||COUNT||N0) as part of set A.

4544 d. TPM exports U.

4545 9. Stage 8

4546 a. TPM imports ENC_NE (2048-bit).

4547 b. TPM decrypts NE (160-bit) from ENC_NE (2048-bit) by using privEK: NE =
4548 decrypt(privEK, ENC_NE).

4549 c. TPM computes U2 (160-bit) = H(U1||NE).

4550 d. TPM erases U1 from set A.

4551 e. TPM exports U2.

4552 10. Stage 9

4553 a. TPM generates a TPM-specific secret r0 (344-bit) from the RNG.

4554 b. TPM generates a TPM-specific secret r1 (344-bit) from the RNG.

4555 c. TPM generates a TPM-specific secret r2 (1024-bit) from the RNG.

4556 d. TPM generates a TPM-specific secret r3 (1264-bit) from the RNG.

4557 e. TPM stores r0, r1, r2, r3 as part of set A.

4558 f. TPM computes a non-secret value P1 (2048-bit) = (R0^r0) mod n and stores P1 as part of
4559 set A.

4560 11. Stage 10

4561 a. TPM computes a non-secret value P2 (2048-bit) = P1*(R1^r1) mod n, stores P2 as part of
4562 set A and erases P1 from set A.

4563 12. Stage 11

4564 a. TPM computes a non-secret value P3 (2048-bit) = P2*(S0^r2) mod n, stores P3 as part of
4565 set A and erases P2 from set A.

4566 13. Stage 12

4567 a. TPM computes a non-secret value P4 (2048-bit) = P3*(S1^r3) mod n, stores P4 as part of
4568 set A and erases P3 from set A.

4569 b. TPM exports P4.

4570 14. Stage 13

4571 a. TPM imports w (2048-bit).

4572 b. TPM computes w1 = w^q mod Γ.

 Copyright © TCG

4573 c. TPM verifies if w1 = 1 holds. If it doesn't hold, TPM refuses to continue.
4574 d. If it does hold, TPM saves w as part of set A.
4575 15. Stage 14
4576 a. TPM computes a non-secret value E (2048-bit) = w^f mod Γ.
4577 b. TPM exports E.
4578 16. Stage 15
4579 a. TPM computes a TPM-specific secret r (208-bit) = r0 + 2^104*r1 mod q.
4580 b. TPM computes a non-secret value E1 (2048-bit) = w^r mod Γ.
4581 c. TPM exports E1 and erases w from set A.
4582 17. Stage 16
4583 a. TPM imports a non-secret value c1 (160-bit).
4584 b. TPM generates a non-secret value NT (160-bit) from the RNG.
4585 c. TPM computes a non-secret value c (160-bit) = H(c1||NT).
4586 d. TPM save c as part of set A.
4587 e. TPM exports NT
4588 18. Stage 17
4589 a. TPM computes a non-secret value s0 (352-bit) = r0 + c*f0 over the integers.
4590 b. TPM exports s0.
4591 19. Stage 18
4592 a. TPM computes a non-secret value s1 (352-bit) = r1 + c*f1 over the integers.
4593 b. TPM exports s1.
4594 20. Stage 19
4595 a. TPM computes a non-secret value s2 (1024-bit) = r2 + c*u0 mod 21024.
4596 b. TPM exports s2.
4597 21. Stage 20
4598 a. TPM computes a non-secret value s'2 (1024-bit) = (r2 + c*u0) >> 1024 over the integers.
4599 b. TPM saves s'2 as part of set A.
4600 c. TPM exports c
4601 22. Stage 21
4602 a. TPM computes a non-secret value s3 (1272-bit) = r3 + cu1 + s'2 over the integers.
4603 b. TPM exports s3 and erases s'2 from set A.
4604 23. Stage 22
4605 a. TPM imports a non-secret value u2 (1024-bit).
4606 b. TPM computes a TPM-specific secret v0 (1024-bit) = u2 + u0 mod 21024.

4607 c. TPM stores v0 as part of A.

4608 d. TPM computes a TPM-specific secret v'0 (1024-bit) = (u2 + u0) >> 1024 over the integers.

4609 e. TPM saves v'0 as part of set A.

4610 24. Stage 23

4611 a. TPM imports a non-secret value u3 (1512-bit).

4612 b. TPM computes a TPM-specific secret v1 (1520-bit) = u3 + u1 + v'0 over the integers.

4613 c. TPM stores v1 as part of A.

4614 d. TPM erases v'0 from set A.

4615 25. Stage 24

4616 a. TPM makes self-consistent set of all the data (n0, COUNT, R0, R1, S0, S1, n, Γ, q, v0,
4617 v1), where the values v0, v1 are secret – they need to be stored safely with the consistent
4618 set, and the remaining is non-secret.

4619 b. TPM erases set A.

4620 **End of informative comment**

4621 33.3.3 SIGN

4622 **Start of informative comment**

4623 This entire section is informative

4624 1. Stage 0 & 1

4625 a. TPM imports and verifies a self-consistent set of all the data including:

4626 i. n0 (2048-bit),

4627 ii. COUNT (8-bit),

4628 iii. R0 (2048-bit),

4629 iv. R1 (2048-bit),

4630 v. S0 (2048-bit),

4631 vi. S1 (2048-bit),

4632 vii. n (2048-bit),

4633 viii. gamma (2048-bit),

4634 ix. q (208-bit),

4635 x. v0 (1024-bit),

4636 xi. v1 (1520-bit).

4637 xii. If the verification does not succeed, TPM refuses to continue.

4638 b. TPM stores the above values as part of a new set A.

4639 c. TPM computes a TPM-specific secret f0 (104-bit) = f mod 2104.

4640 d. TPM computes a TPM-specific secret f1 (104-bit) = f >> 104.

Copyright © TCG

4641 e. TPM stores f0 and f1 as part of set A.

4642 f. TPM generates a TPM-specific secret r0 (344-bit) from the RNG.

4643 g. TPM generates a TPM-specific secret r1 (344-bit) from the RNG.

4644 h. TPM generates a TPM-specific secret r2 (1024-bit) from the RNG.

4645 i. TPM generates a TPM-specific secret r4 (1752-bit) from the RNG.

4646 j. TPM stores r0, r1, r2, r4, as part of set A.

4647 2. Stage 2

4648 a. TPM computes a non-secret value P1 (2048-bit) = (R0^r0) mod n and stores P1 as part of
4649 set A.

4650 3. Stage 3

4651 a. TPM computes a non-secret value P2 (2048-bit) = P1*(R1^r1) mod n, stores P2 as part of
4652 set A and erases P1 from set A.

4653 4. Stage 4

4654 a. TPM computes a non-secret value P3 (2048-bit) = P2*(S0^r2) mod n, stores P3 as part of
4655 set A and erases P2 from set A.

4656 5. Stage 5

4657 a. TPM computes a non-secret value T (2048-bit) = P3*(S1^r4) mod n.

4658 b. TPM erases P3 from set A.

4659 c. TPM exports T.

4660 6. Stage 6

4661 a. TPM imports a non-secret value w (2048-bit).

4662 b. TPM computes w1 = w^q mod Γ.

4663 c. TPM verifies if w1 = 1 holds. If it doesn't hold, TPM refuses to continue.

4664 d. If it does hold, TPM saves w as part of set A.

4665 7. Stage 7

4666 a. TPM computes a non-secret value E (2048-bit) = w^f mod Γ.

4667 b. TPM exports E and erases f from set A.

4668 8. Stage 8

4669 a. TPM computes a TPM-specific secret r (208-bit) = r0 + 2^104*r1 mod q.

4670 b. TPM computes a non-secret value E1 (2048-bit) = w^r mod Γ.

4671 c. TPM exports E1 and erases w and E1 from set A.

4672 9. Stage 9

4673 a. TPM imports a non-secret value c1 (160-bit).

4674 b. TPM generates a non-secret value NT (160-bit) from the RNG.

4675 c. TPM computes a non-secret value c2 (160-bit) = H(c1||NT) and erases c1 from set A.

4676 d. TPM saves c2 as part of set A.

4677 e. TPM exports NT.

4678 10. Stage 10

4679 a. TPM imports a non-secret value b (1-bit).

4680 b. If b = = 1, TPM imports a non-secret value m (160-bit).

4681 c. TPM computes a non-secret value c (160-bit) = H(c2||b||m) and erases c2 from set A.

4682 d. If b = = 0, TPM imports an RSA public key, eAIK (= 2^16 + 1) and nAIK (2048-bit).

4683 e. TPM computes a non-secret value c (160-bit) = H(c2||b||nAIK) and erases c2 from set
4684 A.

4685 f. TPM exports c.

4686 11. Stage 11

4687 a. TPM computes a non-secret value s0 (352-bit) = r0 + c*f0 over the integers.

4688 b. TPM exports s0.

4689 12. Stage 12

4690 a. TPM computes a non-secret value s1 (352-bit) = r1 + c*f1 over the integers.

4691 b. TPM exports s1.

4692 13. Stage 13

4693 a. TPM computes a non-secret value s2 (1024-bit) = r2 + c*v0 mod 21024.

4694 b. TPM exports s2.

4695 14. Stage 14

4696 a. TPM computes a non-secret value s'2 (1024-bit) = (r2 + c*v0) >> 1024 over the integers.

4697 b. TPM saves s'2 as part of set A.

4698 15. Stage 15

4699 a. TPM computes a non-secret value s3 (1760-bit) = r4 + cv1 + s'2 over the integers.

4700 b. TPM exports s3 and erases s'2 from set A.

4701 c. TPM erases set A.

4702 **End of informative comment**

Copyright © TCG

4703 # 34. General Purpose IO

4704 **Start of informative comment**

4705 The GPIO capability allows an outside entity to output a signal on a GPIO pin, or read the
4706 status of a GPIO pin. The solution is for a single pin, with no timing information. There is
4707 no support for sending information on specific busses like SMBus or RS232. The design
4708 does support the designation of more than one GPIO pin.

4709 There is no requirement as to the layout of the GPIO pin, or the routing of the wire from the
4710 GPIO pin on the platform. A platform specific specification can add those requirements.

4711 To avoid the designation of additional command ordinals, the architecture uses the NV
4712 Storage commands. A set of GPIO NV indexes map to individual GPIO pins.
4713 TPM_NV_INDEX_GPIO_00 maps to the first GPIO pin. The platform specific specification
4714 indicates the mapping of GPIO zero to a specific package pin.

4715 The TPM does not reserve any NV storage for the indicated pin; rather the TPM uses the
4716 authorization mechanisms for NV storage to allow a rich set of controls on the use of the
4717 GPIO pin. The TPM owner can specify when and how the platform can use the GPIO pin.
4718 While there is no NV storage for the pin value, TRUE or FALSE, there is NV storage for the
4719 authorization requirements for the pin.

4720 Using the NV attributes the GPIO pin may be either an input pin or an output pin.

4721 **End of informative comment**

4722 1. The TPM MAY support the use of a GPIO pin defined by the NV storage mechanisms.

4723 2. The GPIO pin MAY be either an input or an output pin.

4724 # 35. Redirection

4725 **Informative comment**

4726 Redirection allows the TPM to output the results of operations to hardware other than the
4727 normal TPM communication bus. The redirection can occur to areas internal or external to
4728 the TPM. Redirection is only available to key operations (such as TPM_UnBind,
4729 TPM_Unseal, and TPM_GetPubKey). To use redirection the key must be created specifying
4730 redirection as one of the keys attributes.

4731 When redirecting the output the TPM will not interpret any of the data and will pass the
4732 data on without any modifications.

4733 The TPM_SetRedirection command connects a destination location or port to a loaded key.
4734 This connection remains so long as the key is loaded, and is saved along with other key
4735 information on a saveContext(key), loadContext(key). If the key is reloaded using
4736 TPM_LoadKey, then TPM_SetRedirection must be run again.

4737 Any use of TPM_SetRedirection with a key that does not have the redirect attribute must
4738 return an error. Use of key that has the redirect attribute without TPM_SetRedirection being
4739 set must return an error.

4740 **End of informative comments**

4741 1. The TPM MAY support redirection

4742 2. If supported, the TPM MUST only use redirection on keys that have the redirect attribute
4743 set

4744 3. A key that is tagged as a “redirect” key MUST be a leaf key in the TPM Protected Storage
4745 blob hierarchy. A key that is tagged as a “redirect” key CAN NEVER be a parent key.

4746 4. Output data that is the result of a cryptographic operation using the private portion of a
4747 “redirect” key:

4748 a. MUST be passed to an alternate output channel

4749 b. MUST NOT be passed to the normal output channel

4750 c. MUST NOT be interpreted by the TPM

4751 5. When command input or output is redirected the TPM MUST respond to the command
4752 as soon as the ordinal finishes processing

4753 a. The TPM MUST indicate to any subsequent commands that the TPM is busy and
4754 unable to accept additional command until the redirection is complete

4755 b. The TPM MUST allow for the resetting of the redirection channel

4756 6. Redirection MUST be available for the following commands:

4757 a. TPM_Unseal

4758 b. TPM_UnBind

4759 c. TPM_GetPubKey

4760 d. TPM_Seal

4761 e. TPM_Quote

4762 # 36. Structure Versioning

4763 **Start of informative comment**

4764 In version 1.1 some structures also contained a version indicator. The TPM set the indicator
4765 to indicate the version of the TPM that was creating the structure. This was incorrect
4766 behavior. The functionality of determining the version of a structure is radically different in
4767 1.2.

4768 Most structures will contain a TPM_STRUCTURE_TAG. All future structures must contain
4769 the tag, the only structures that do not contain the tag are 1.1 structures that are not
4770 modified in 1.2. This restriction keeps backwards compatibility with 1.1.

4771 Any 1.2 structure must not contain a 1.1 tagged structure. For instance the TPM_KEY
4772 complex, if set at 1.2, must not contain a PCR_INFO structure. The TPM_KEY 1.2 structure
4773 must contain a PCR_INFO_LONG structure. The converse is also true 1.1 structures must
4774 not contain any 1.2 structures.

4775 The TPM must not allow the creation of any mixed structures. This implies that a command
4776 that deals with keys, for instance, must ensure that a complete 1.1 or 1.2 structure is
4777 properly built and validated on the creation and use of the key.

4778 The tag structure is set as a UINT16. This allows for a reasonable number of structures
4779 without wasting space in the buffers.

4780 To obtain the current TPM version the caller must use the TPM_GetCapability command.

4781 The tag is not a complete validation of the validity of a structure. The tag provides a
4782 reference for the structure and the TPM or caller is responsible for determining the validity
4783 of any remaining fields. For instance, in the TPM_KEY structure, the tag would indicate
4784 TPM_KEY but the TPM would still use tpmProof and the various digests to ensure the
4785 structure integrity.

4786 7. Compatibility and notification

4787 In 1.1 TPM_CAP_VERSION (index 19) returned a version structure with 1.1.x.x. The x.x was
4788 for manufacturer information and the x.x also was set version structures. In 1.2
4789 TPM_CAP_VERSION will return 1.1.0.0. Any 1.2 structure that uses the version information
4790 will set the x.x to 0.0 in the structure. TPM_CAP_MANUFACTURER_VER (index 21) will
4791 return 1.2.x.x. The 1.2 structures do not contain the version structure. The rationale
4792 behind this is that the structure tag will indicate the version of the structure. So changing a
4793 correct structure will result in a new tag and there is no need for a separate version
4794 structure.

4795 For further compatibility, the quote function always returns 1.1.0.0 in the version
4796 information regardless of the size of the incoming structure. All other functions may regard
4797 a 2 byte sizeofselect structure as indicative of a 1.1 structure. The TPM handles all of the
4798 structures according to the input, the only exception being TPM_CertifyKey where the TPM
4799 does not need to keep the input version of the structure.

4800 **End of informative comment**

4801 1. The TPM MUST support 1.1 and 1.2 defined structures

4802 2. The TPM MUST ensure that 1.1 and 1.2 structures are not mixed in the same overall
4803 structure

4804 a. For instance in the TPM_KEY structure if the structure is 1.1 then PCR_INFO MUST
4805 be set and if 1.2 the PCR_INFO_LONG structure must be set

4806 3. On input the TPM MUST ignore the lower two bytes of the version structure

4807 4. On output the TPM MUST set the lower two bytes to 0 of the version structure

 Copyright © TCG

37. Certified Migration Key Type

Start of informative comment

In version 1.1 there were two key types, non-migration and migration keys. The TPM would only certify non-migrating keys. There is a need for a key that allows migration but allows for certification. This proposal is to create a key that allows for migration but still has properties that the TPM can certify.

These new keys are "certifiable migratable keys" or CMK. This designation is to separate the keys from either the normal migration or non-migration types of keys. The TPM Owner is not required to use these keys.

Two entities may participate in the CMK process. The first is the Migration-Selection Authority and the second is the Migration Authority (MA).

Migration Selection Authority (MSA)

The MSA controls the migration of the key but does not handle the migrated itself.

Migration Authority (MA)

A Migration Authority actually handles the migrated key.

Use of MSA and MA

Migration of a CMK occurs using TPM_CMK_CreateBlob (TPM_CreateMigrationBlob cannot be used). The TPM Owner authorizes the migration destination (as usual), and the key owner authorizes the migration transformation (as usual). An MSA authorizes the migration destination as well. If the MSA is the migration destination, no MSA authorization is required.

End of informative comment

37.1 Certified Migration Requirements

Start of informative comment

The following list details the design requirements for the controlled migration keys

Key Protections

The key must be protected by hardware and an entity trusted by the key user.

Key Certification

The TPM must provide a mechanism to provide certification of the key protections (both hardware and trusted entity)

Owner Control

The TPM Owner must control the selection of the trusted entity

Control Delegation

The TPM Owner may delegate the ability to create the keys but the decision must be explicit

Linkage

The architecture must not require linking the trusted entity and the key user

4844 **Key Type**

4845 The key may be any type of migratable key (storage or signing)

4846 **Interaction**

4847 There must be no required interaction between the trusted entity and the TPM during the
4848 key creation process

4849 **End of informative comment**

4850 37.2 Key Creation

4851 **Start of informative comment**

4852 The command TPM_CMK_CreateKey creates a CMK where control of the migration is by a
4853 MSA or MA. The process uses the MSA public key (actually a digest of the MA public key) as
4854 input to TPM_CMK_CreateKey. The key creation process establishes a migrationAuth that is
4855 SHA-1(tpmProof || SHA-1(MA pubkey) || SHA-1(source pubkey)).

4856 The use of tpmProof is essential to prove that CMK creation occurs on a TPM. The use of
4857 “source pubkey” explicitly links a migration AuthData value to a particular public key, to
4858 simplify verification that a specific key is being migrated.

4859 **End of informative comment**

4860 37.3 Migrate CMK to a MA

4861 **Start of informative comment**

4862 Migration of a CMK to a destination other than the MSA:

4863 **TPM_MIGRATIONKEYAUTH Creation**

4864 The TPM Owner authorizes the creation of a TPM_MIGRATIONKEYAUTH structure using
4865 TPM_AuthorizeMigrationKey command. The structure contains the destination
4866 migrationKey, the migrationScheme (which must be set to TPM_MS_RESTRICT_MIGRATE
4867 or TPM_MS_RESTRICT_APPROVE) and a digest of tpmProof.

4868 **MA Approval**

4869 The MA signs a TPM_CMK_AUTH structure, which contains the digest of the MA public key,
4870 the digest of the destination (or parent) public key and a digest of the public portion of the
4871 key to be migrated

4872 **TPM Owner Authorization**

4873 The TPM Owner authorizes the MA approval using TPM_CMK_CreateTicket and produces a
4874 signature ticket

4875 **Key Owner Authorization**

4876 The CMK owner passes the TPM Owner MA authorization, the MSA Approval and the
4877 signature ticket to the TPM_CMK_CreateBlob using the key owners authorization.

4878 Thus the TPM owner, the key’s owner, and the MSA, all cooperate to migrate a key
4879 produced by TPM_CMK_CreateBlob.

4880 **End of informative comment**

 Copyright © TCG

4881 ## 37.4 Migrate CMK to a MSA

4882 **Start of informative comment**

4883 Migrate CMK directly to a MSA

4884 **TPM_MIGRATIONKEYAUTH Creation**

4885 The TPM Owner authorizes the creation of a TPM_MIGRATIONKEYAUTH structure using
4886 TPM_AuthorizeMigrationKey command. The structure contains the destination
4887 migrationKey (which must be the MSA public key), the migrationScheme (which must be set
4888 to TPM_MS_RESTRICT_MIGRATE) and a digest of tpmProof.

4889 **Key Owner Authorization**

4890 The CMK owner passes the TPM_MIGRATIONKEYAUTH to the TPM in a
4891 TPM_CMK_CreateBlob using the CMK owner authorization.

4892 **Double Wrap**

4893 If specified, through the MS_MIGRATE scheme, the TPM double wraps the CMK information
4894 such that the only way a recipient can unwrap the key is with the cooperation of the CMK
4895 owner.

4896 **Proof of Control**

4897 To prove to the MA and to a third party that migration of a key is under MSA control, a
4898 caller passes the MA's public key (actually its digest) to TPM_CertifyKey, to create a
4899 TPM_CERTIFY_INFO structure. This now contains a digest of the MA's public key.

4900 A CMK be produced without cooperation from the MA: the caller merely provides the MSA's
4901 public key. When the restricted key is to be migrated, the public key of the intended
4902 destination, plus the CERTIFY_INFO structure are sent to the MSA. The MSA extracts the
4903 migrationAuthority digest from the CERTIFY_INFO structure, verifies that
4904 migrationAuthority corresponds to the MSA's public key, creates and signs a
4905 TPM_RESTRICTEDKEYAUTH structure, and sends that signature back to the caller. Thus
4906 the MSA never needs to touch the actual migrated data.

4907 **End of informative comment**

38. Revoke Trust

Start of informative comment

There are circumstances where clearing all keys and values within the TPM is either desirable or necessary. These circumstances may involve both security and privacy concerns.

Platform trust is demonstrated using the EK Credential, Platform Credential and the Conformance Credentials. There is a direct and cryptograph relationship between the EK and the EK Credential and the Platform Credential. The EK and Platform credentials can only demonstrate platform trust when they can be validated by the Endorsement Key.

This command is called revoke trust because by deleting the EK, the EK Credential and the Platform Credential are dissociated from platform therefore invalidating them resulting in the revocation of the trust in the platform. From a trust perspective, the platform associated with these specific credentials no longer exists. However, any transaction that occurred prior to invoking this command will remain valid and trusted to the same extent they would be valid and trusted if the platform were physically destroyed.

This is a non-reversible function. Also, along with the EK, the Owner is also deleted removing all non-migratable keys and owner-specified state.

It is possible to establish new trust in the platform by creating a new EK using the TPM_CreateRevocableEK command. (It is not possible to create an EK using the TPM_CreateEndorsementKeyPair because that command is not allowed if the revoke trust command is allowed.) Establishing trust in the platform, however, is more than just creating the EK. The EK Credential and the Platform Credential must also be created and associated with the new EK as described above. (The conformance credentials may be obtained from the TPM and Platform manufacturer.) These credentials must be created by an entity that is trusted by those entities interested in the trust of the platform. This may not be a trivial task. For example, an entity willing to create these credentials my want to examine the platform and require physical access during the new EK generation process.

Besides calling one of the two EK creation functions to create the EK, the EK may be "squirted" into the TPM by an external source. If this method is used, tight controls must be placed on the process used to perform this function to prevent exposure or intentional duplication of the EK. Since the revocation and re-creation of the EK are functions intended to be performed after the TPM leaves the trusted manufacturing process, squiring of the EK must be disallowed if the revoke trust command is executed.

End of informative comment

1. The TPM MUST not allow both the TPM_CreateRevocableEK and the TPM_CreateEndorsementKeyPair functions to be operational.
2. After an EK is created the TPM MUST NOT allow a new EK to be "squirted" for the lifetime of the TPM.
3. The EK Credential MUST provide an indication within the EK Credential as to how the EK was created. The valid permutations are:
   a. Squirted, non-revocable
   b. Squirted, revocable
   c. Internally generated, non-revocable

Copyright © TCG

4951 d. Internally generated, revocable

4952 4. If the method for creating the EK during manufacturing is squiring the EK may be either
4953 non-revocable or revocable. If it is revocable, the method must provide the insertion or
4954 extraction of the EKreset value.

4955 # 39. Mandatory and Optional Functional Blocks

4956 **Start of informative comment**

4957 This section lists the main functional blocks of a TPM (in arbitrary order), states whether
4958 that block is mandatory or optional in the main TPM specification, and provides brief
4959 justification for that choice.

4960 Important notes:

4961 1. The default classification of a TPM function block is “mandatory”, since reclassification
4962 from mandatory to optional enables the removal of a function from existing
4963 implementations, while reclassification from optional to mandatory may require the addition
4964 of functionality to existing implementations.

4965 2. Mandatory functions will be reclassified as optional functions if those functions are not
4966 required in some particular type of TCG trusted platform.

4967 3. If a functional block is mandatory in the main specification, the functionality must be
4968 present in all TCG trusted platforms.

4969 4. If a functional block is optional in the main specification, each individual platform-
4970 specific specification must declare the status of that functionality as either (1) “mandatory-
4971 specific” (the functionality must be present in all platforms of that type), or (2) “optional-
4972 specific” (the functionality is optional in that type of platform), or (3) “excluded-specific” (the
4973 functionality must not be present in that type of platform).

4974 **End of informative comment**

4975 Classification of TPM functional blocks

4976 1. Legacy (v1.1b) features

4977 a. Anything that was mandatory in v1.1b continues to be mandatory in v1.2. Anything
4978 that was optional in v1.1b continues to be optional in v1.2.

4979 b. V1.2 must be backwards compatible with v1.1b. All TPM features in v1.1b were
4980 discussed in depth when v1.1b was written, and anything that wasn't thought
4981 strictly necessary was tagged as "optional".

4982 2. Number of PCRs

4983 a. The platform specific specification controls the number of PCR on a platform. The
4984 TPM MUST implement the mandatory number of PCR specified for a particular
4985 platform

4986 i. TPMs designed to work on multiple platforms MUST provide the appropriate
4987 number of TPM for all intended platforms. I.e. if one platform requires 16 PCR
4988 and the other platform 24 the TPM would have to supply 24 PCR.

4989 b. For TPMs providing backwards compatibility with 1.1 TPM on the PC platform, there
4990 MUST be 16 static PCR.

4991 3. Sessions

4992 a. The TPM MUST support a minimum of 3 active sessions

4993 i. Active means currently loaded and addressable inside the TPM

4994 ii. Without 3 active sessions many TPM commands cannot function

4995 b. The TPM MUST support a minimum of 16 concurrent sessions

4996 i. The contextList of currently available session has a minimum size of 16

4997 ii. Providing for more concurrent sessions allows the resource manager additional
4998 flexibility and speed

4999 4. NVRAM

5000 a. There are 20 bytes mandatory of NVRAM in v1.2 as specified by the main
5001 specification. A platform specific specification can require a larger amount of NVRAM

5002 b. Cost is important. The mandatory amount of NVRAM must be as small as possible,
5003 because different platforms will require different amounts of NVRAM. 20 bytes are
5004 required for (DIR) backwards compatibility with v1.1b.

5005 5. New key types

5006 a. The new signing keys are mandatory in v1.2 because they plug a security hole.

5007 6. Direct Anonymous Attestation

5008 a. This is optional in v1.2

5009 b. Cost is important. The DAA function consumes more TPM resources than any other
5010 TPM function, but some platform specific specifications (some servers, for example)
5011 may have no need for the anonymity and pseudonymity provided by DAA.

5012 7. Transport sessions

5013 a. These are mandatory in v1.2.

5014 b. Transport sessions

5015 i. Enable protection of data submitted to a TPM and produced by a TPM

5016 ii. Enable proof of the TPM commands executed during an arbitrary session.

5017 8. Resettable Endorsement Key

5018 a. This is optional in v1.2

5019 b. Cost is important. Resettable EKs are valuable in some markets segments, but cause
5020 more complexity than non-resettable EKs, which are expected to be the dominant
5021 type of EK

5022 9. Monotonic Counter

5023 a. This is mandatory in v1.2

5024 b. A monotonic counter is essential to enable software to defeat certain types of attack,
5025 by enabling it to determine the version (revision) of dynamic data.

5026 10. Time Ticks

5027 a. This is mandatory in v1.2

5028 b. Time stamping is a function that is potentially beneficial to both a user and system
5029 software.

5030 11. Delegation (includes DSAP)

5031 a. This is mandatory in v1.2

5032 b. Delegation enables the well-established principle of least privilege to be applied to
5033 Owner authorized commands.

5034 12. GPIO

5035 a. This is optional in v1.2

5036 b. Cost is important. Not all types of platform will require a secure intra-platform
5037 method of key distribution

5038 13. Locality

5039 a. The use of locality is optional in v1.2

5040 b. The structures that define locality are mandatory

5041 c. Locality is an essential part of many (new) TPM commands, but the definition of
5042 locality varies widely from platform to platform, and may not be required by some
5043 types of platforms.

5044 d. It is mandatory that a platform specific specification indicate the definitions of
5045 locality on the platform. It is perfectly reasonable to only define one locality and
5046 ignore all other uses of locality on a platform

5047 14. TPM-audit

5048 a. This is optional in v1.2

5049 b. Proper TPM-audit requires support to reliably store logs and control access to the
5050 TPM, and any mechanism (an OS, for example) that could provide such support is
5051 potentially capable of providing an audit log without using TPM-audit. Nevertheless,
5052 TPM-audit might be useful to verify operation of any and all software, including an
5053 OS. TPM-audit is believed to be of no practical use in a client, but might be valuable
5054 in a server, for example.

5055 15. Certified Migration

5056 a. This is optional in v1.2

5057 b. Cost is important. Certified Migration enables a business model that may be
5058 nonsense for some platforms.

Copyright © TCG

5059

40. 1.1a and 1.2 Differences

Start of informative comment

All 1.2 TPM commands are completely compliant with 1.1b commands with the following known exceptions.

1. TSC_PhysicalPresence does not support configuration and usage in a single step.

2. TPM_GetPubKey is unable to read the SRK unless TPM_PERMANENT_FLAGS -> readSRKPub is TRUE

3. TPM_SetTempDeactivated now requires either physical presence or TPM Operator authorization to execute

4. TPM_OwnerClear does not modify TPM_PERMANENT_DATA -> authDIR[0].

End of informative comment