TRUSTED® COMPUTING GROUP

R E F E R E N C E

TCG Guidance on Integrity Measurements and Event Log Processing

Version 1.0

Revision 0.118

December 15, 2021

Contact: admin@trustedcomputinggroup.org

PUBLIC REVIEW

Work in Progress

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

DISCLAIMERS, NOTICES, AND LICENSE TERMS

CHANGE HISTORY

REVISION	DATE	DESCRIPTION
1.00/0.10	2019-10-23	Initial Draft of Version 1.00.
1.00/0.11	2019-11-08	Initial Draft for review
1.00/0.80	2019-11-18	Revised per IWG comments
1.00/0.85	2019-11-19	Cleaned up figures; Added text to scope; Added description for PC Client Event Log Record
1.00/0.90	2019-12-10	Incorporated Graeme's comments
1.00/0/91	2020-01-22	Incorporated Ned's and Lawrence's comments
1.00/.100	2020-07-22	Added PCClient discussion and examples
1.00/101	2020-07-29	Clarifications in PCClient discussion
1.00/102	2020-08-05	Clarifications based on comments
1.00/103	2020-08-19	Fixed Figure 9 text
1.00/104	2020-08-26	Describe Collection in section 4 and update section 7 with new log.
1.00/105	2020-09-02	Clarified Attestation Figures and corresponding text, Added Ref 7, overall review
1.00/106	2020-09-09	Added details on appraiser steps to both section 4 and 7.
1.00/107	2020-30-09	Incorporated recommended changes/extensions to section 7
1.00/108	2020-07-10	Rewrites to section 7
1.00/109	2021-05-12	Added CEL references and general cleanup
1.00/110	2021-06-09	Cleanup for review
1.00/111	2021-07-05	Corrected guidance on Quote verification
1.00/112	2021-08-31	Included discussion on verification with PCR list
1.00/113	2021-09-28	Included discussion of audit session-based attestation
1.00/114	2021-11-02	Removing DRTM and boot loader events from scope
1.00/115	2021-11.03	Added legend for Appendix A listing
1.00/116	2021-12-07	Make changes per comments from the Technical Committee
1.00/117-118	2021-12-15	Complete changes per comments from the Technical Committee

Contents

1 Scope and Background

1.1 Scope

This document is guidance for and a companion to the “PC Client Platform Firmware Integrity Measurement Specification” also referred to as FIM [5] and the “TCG Reference Integrity Manifest (RIM) Information Model” [6].

This document explains the concepts and usage of Integrity Measurements (IM), also referred to as Event Log Records, for attestation. This guidance’s focus is the use of Integrity Measurements described in the above mentioned TCG Specifications. This focus includes firmware-based measurements. The scope does not include Dynamic Root of Trust for Measurement (DRTM) boot loaders, such as tboot, operating system boot loaders, such as GRUB, and operating system measurements, such as Linux’s Integrity Measurement Architecture (IMA). This guidance provides implementers an understanding of the concepts and implementation details behind attestation described by the mentioned TCG specifications. Implementers of these specifications include those who develop modules or applications that create the Integrity Measurements (and their associated Event Log Records) such as boot firmware; create the Reference Integrity Measurements (RIMs); develop Verifiers; and those who create policies. Developers of attestation protocols may also benefit from this guidance. A description of concepts and actions after the Verifier’s appraisal step, such as recovery and remediation, is outside the scope of this guidance[1].

This document’s focus is processing Integrity Measurements using the TPM, either Family 1.2[2] or Family 2.0. For this guidance there are few differences between the TPM Families. Any differences will be specifically stated.

1.2 Background

This document’s primarily references the TCG TNC architecture as described in the *TCG Trusted Network Connect TNC Architecture for Interoperability* [1]. However, some of the necessary aspects are defined by TCG platform specific firmware profiles such as the PC Client Platform Firmware Profile [3] that define the Event Log structure. This document will merge the concepts defined by each set of specifications. This document will specifically reference the PC Client Platform Firmware Profile as that is widely deployed. However, the concepts and processes described in this document should equally apply to any platform class.

[1] Those concepts, if provided, may be either an addition to this document in the form a new version or may be a different document.
[2] While all TPM Family 1.2 concepts may also be applied to Family 1.1b, TPM Families prior to 1.2 are out of scope for this document.

2 Concepts and Terminology

2.1 Purpose

The purpose of attestation is the provision of evidence about the mutable components that ran or are running in a platform. A platform's behavior is determined by both its set of immutable components and its set of mutable components. Immutable components are physically unchangeable components such as hardware. Mutable components are the set of components that can change either during manufacturing, later in the supply chain, or when the platform is in possession of the user. Examples of mutable components include boot firmware, pre-OS modules, the OS itself, and loadable drivers and applications. Changing mutable components may result in changing the platform's behavior even without changing the immutable components (i.e., the hardware). While methods such as "secure update" and digitally signing of mutable components provide some protection against unauthorized changes, this is often insufficient. Secure update methods do not provide strong evidence of the platform's behavior between versions of component signed using the same signing key. Anti-rollback methods can be used when updating components to address vulnerabilities, however, there is no strong evidence of which version is currently running. Secure update also provides no protection in cases where the signing key is compromised.

Evidence of the mutable components is provided by creating Integrity Measurements of the mutable components. A series of Integrity Measurements creates a type of audit log of the mutable components. Not all mutable components affect the platform's behavior in a way that affects the platform's trustworthiness. TCG's platform specific specifications (such as the PC Client, Server, Virtualization) identify the set of these mutable components, called Events, which may affect the platform trustworthiness.

2.2 Terminology and Attestation Workflow

This section describes the terminology used in TCG attestation.

2.2.1 Integrity Measurement

From TNC Architecture [1]: "Integrity measurements are used as evidence of the security posture of the endpoint so access control solutions can evaluate the endpoint's suitability for being given access to the network."

Similarly, From TPM 2 Specification Part 1; 9.5.5 [2]:

> "An integrity measurement is a value that represents a possible change in the trust state of the platform. The measured object may be anything of meaning but is often
>
> - a data value,
> - the hash of code or data, or
> - an indication of the signer of some code or data."

An Integrity Measurement (often called Measurement or IM for brevity) is a recording of an Event as defined above. These Integrity Measurements are used along with a policy to appraise the platform's[3] trust state.

2.2.2 Roles and Interactions

While other TCG documents provide a more detailed description of the roles involved in attestation this document will provide an overview of these role and how they interact with other roles within an attestation framework.

Assertion: An expected value such as a PCR, Integrity Measurement, or Event Record from a trusted entity.

Attester: From TAP [4]: "A platform or platform component that provides evidence to a Verifier as to its state."

Verifier: From TAP [4]: A system that analyzes evidence from an Attester to determine the Attester's state.

Asserter: From RIM-IM [6]: A supply chain entity, manufacturer, vendor, or reseller that produces reference values.

[3] For this document, a platform is something which has an associated Root of Trust for Reporting. Examples include a physical device such as a PC Client and a Virtual Platform running on a hypervisor.

2.3 Terminology used in this Document

There are a variety of terms used for attestation that are defined in depth later in this document, Table 1 provides a summary.

Term	Definition
Event	Executable, Data, or action that may affect the device's trust state
Event Log	A set of related Event Log Records
Event Log Record	An IM and bound set of meta-data about the Event
Event Record	Same as Event Record
Integrity Measurement (IM)	A deterministic (1:1) representation of an Event
Integrity Measurement Log (IML)	A set of either IMs or Event Log Records. This is also known as the Event Log in some TCG documents. These are synonymous.
Reference Integrity Manifest (RIM)	A Reference Integrity Manifest contains data such as expected PCR values or expected measurements that a Verifier uses to validate expected values (Assertions) against actual values (Evidence).
PCR Value	The result of the extension of Integrity Measurements into a PCR.
PCR Value Verification	Verification of integrity status based on Reference (known good) PCR Values
Event Log Verification	Verification of integrity status based on verification of all Event Log Records against Reference (known good) values
Quote	Signing of attestation data (PCR values or hashes of PCR values), made by a TPM using a key trusted by the verifier. Specific Quote methods include TPM_Quote, TPM2_Quote, and using a signed audit session. These methods are discussed in detail in section 4.

Table 1 Terminology

3 Integrity Measurement Collection

3.1 Process

3.1.1 Creating Integrity Measurements

Events may be of arbitrary size. Some Events may be large blocks of executable code (greater than several megabytes) and other Events may be small strings such as a version number, configuration data, etc. For efficient storage, transmission and appraisal, an entire Event is transformed into a standardized representation of that Event. This representation is called an Integrity Measurement. This is depicted in Figure 1.TCG uses the size of a hash to hold an Integrity Measurement. A critical property of this transformation is the one-to-one relationship between the Event and its associated Integrity Measurement. Because TCG uses a digest size value to represent an Integrity Measurement, the term digest is often used in TCG documents as a synonym for “Integrity Measurement. These terms are often interchangeable.

For example, if the Event is a large binary object (such firmware executable code) the Integrity Measurement will be a cryptographic hash of that Event resulting in a digest-sized Integrity Measurement. Another example is a version string of an executable such as an RTM[4] where the version string is less that the size of the digest. The version string is padded, resulting a digest-sized Integrity Measurement of the version string.

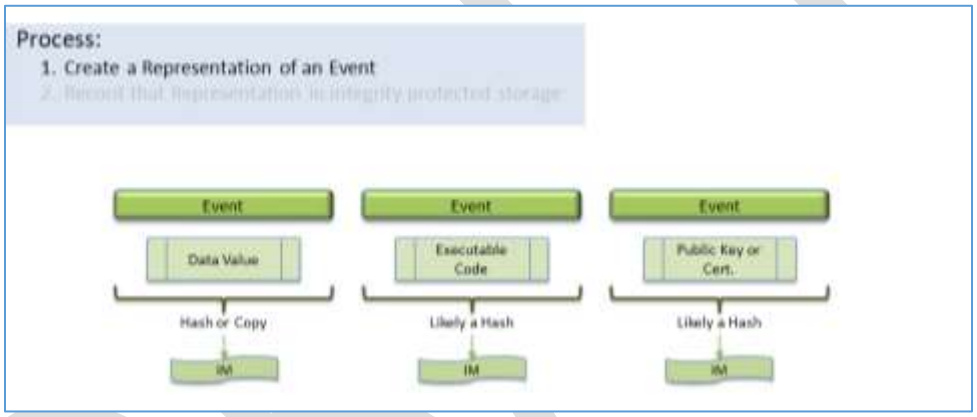


Figure 1 Measurement Process

[4] In this example, if the executable is the RTM, there is no security value is creating a hash of an RTM, since Roots of Trust are trusted to behave as expected. However, there may be value in the verifier knowing the version of the RTM. While the RTM could create a digest of itself, that would just waste time during the time-critical boot phase of a platform, so the PC Client specification only requires measuring the RTM’s version string.

3.1.2 Recording Integrity Measurements

After an Integrity Measurement is created, it must be stored in an integrity protected location[5]. While reduced in size, there are typically too many Integrity Measurements to store directly into the TPM. For example, a PC Client's firmware may perform separate measurements for just the early stages of the boot process. The TPM therefore, has an Extend operation, shown in Figure 2, allowing an accumulation of Integrity Measurements to be recorded.

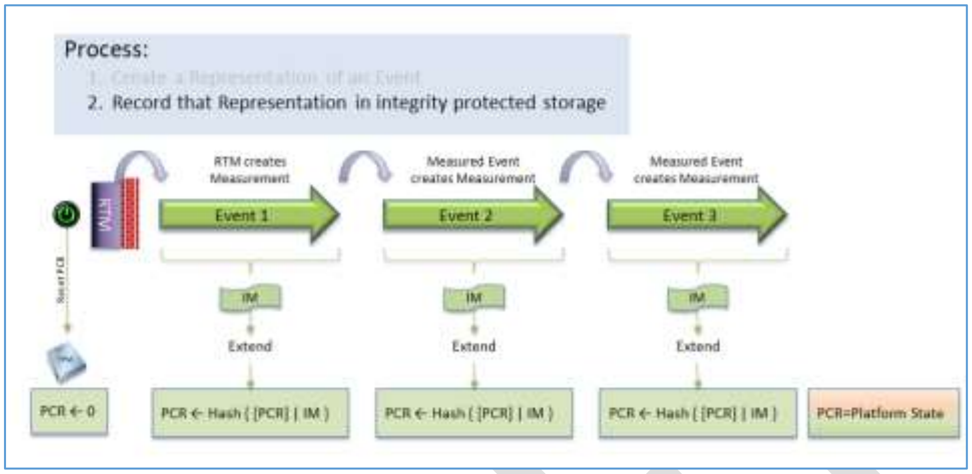

Process:
1. Create a Representation of an Event
2. Record that Representation in integrity protected storage
RTM creates Measurement
Measured Event creates Measurement
Measured Event creates Measurement
RTM
Event 1
Event 2
Event 3
Reset PCR
IM
IM
IM
Extend
Extend
Extend
PCR ← 0
PCR ← Hash ([PCR] | IM)
PCR ← Hash ([PCR] | IM)
PCR ← Hash ([PCR] | IM)
PCR=Platform State

Figure 2 Measurement Process: PCR Composite

The Extend operation operates on specific objects within the TPM that are exclusively dedicated to the accumulation of Integrity Measurements. The most familiar object for this purpose is called a Platform Configuration Register or PCR. A TPM typically contains more than one PCR but for simplicity, most of this explanation only one PCR is used. TPM Family 2.0 added a new type of object with similar properties to the PCR called an Extend NV Index[6].

3.1.3 Integrity Measurement Sequence

Due to the nature of the Extend's hashing operation, the order in which the Integrity Measurements are recorded is critical. Events may be recorded in different order due to multiprocessing, different option card placement, etc. For example: if the following Events occurred:

A   B   C

with the resulting Integrity Measurements:

A_IM   B_IM   C_IM

If the recording of the Integrity Measurements were in this order:

A_IM   B_IM   C_IM: The result would be a specific PCR Value: PCR_1

If these same Events were recorded in a different order:

B_IM   A_IM   C_IM: The result would be a different specific PCR Value: PCR_2

The PCR Values are different even though the same Events were involved. Depending on policy, the recording order may be critical to the platform trust state.

[5] Measurements must be protected from the recorded Events. Allowing the recorded events to alter the measurement would negate the value of creating the Integrity Measurements and the verifier could not trust them.
[6] For this guidance, the properties of an Extend NV Index and PCR are equivalent, therefore the term PCR will be used, but the concepts and processes apply equally to both.

3.1.4 Integrity Measurement Concurrency[7]

The Extend operation and the placement of the Integrity Measurement (IM) within an Integrity Measurement Log (IML) are separate operations. The Verifier has a complete set of IMs to compare with the PCR if there is no request for the IML between the Extend operation and placing new IM into the IML. This would be the normal case when multiprocessing is not occurring such as firmware boot operations. However, there are times, especially during runtime (e.g., Linux IMA[8]), where an Attestation request may be serviced between the Extend operation and the recording of the Integrity Measurement. If this occurs the Attestation request may contain a different number of Integrity Measurement Records than was Extended.

If the Attestation request is serviced after the Extend operation but before the Integrity Measurement Record is recorded, the Verifier will have an incomplete Integrity Measurement Log and will not be able to verify any of the Integrity Measurement Records.

If the Attestation request is serviced after the Integrity Measurement Record is recorded but before the Extend operation, the Verifier will have a complete Integrity Measurement Log (plus more) and will be able to verify the Integrity Measurement Records by discarding the more recent Integrity Measurement Records.

Attesters where the above cases may be relevant should be designed to permit Verifiers to perform their functions. Techniques such as always recording Integrity Measurement Records before the Extend operation or using synchronization techniques to prevent Attestation operations from being serviced between measurement operations should be considered. Verifiers should be designed with the above considerations in mind.

3.2 Integrity Measurement: PCR Value

The result of the collection of the Integrity Measurements is a PCR Value. This value represents the collection of all Integrity Measurements relevant to the platform's trust state up to the point the PCR is read. It represents the composite state of the platform. Due to the nature of the Extend process (i.e., the Pre-image resistant nature of the hash function) it is infeasible to obtain any information about individual Integrity Measurements within the PCR. The value of a PCR is therefore a Composite of the Events and there is no method for extracting any individual Integrity Measurement from the value of the PCR.

3.3 Integrity Measurement: Event Log

3.3.1 Benefit of an Integrity Measurement Log (or Event Log)

There may be situations where there is benefit in having access to individual Integrity Measurements. One example is that while the order of the Integrity Measurements may not be relevant to the system's trust state, the PCR's composite value is order dependent as described above. In this case, if the Verifier had the individual expected Integrity Measurements (called Assertions), it could simply change the order during the appraisal. Another example is that some policies may permit a set of optional Events. In yet another example, a single Event has a wide variety of options (such as configuration settings). In this case, only a set of the permutations of that single Event would need to be known. Without having the set of the individual Integrity Measurements, called an Integrity Measurement Log (or Event Log), every permutation of PCR Value would have to be known.

3.3.2 Recording Integrity Measurements within an Integrity Measurement Log

Figure 3 shows how retaining each Integrity Measurement is as simple as copying the Integrity Measurement which is Extended into a storage location. As the Integrity Measurement for each Event is already secured by the TPM using the Extend process, there is no need for additional protection for each Integrity Measurement while in storage. The storage location may be anywhere including memory, the filesystem, or event transferred off the system. The only requirement is, as noted in section above, that the order of the Integrity Measurements is maintained. It should be noted, however, that the only attack known for changing the order of the Integrity Measurement is a denial of service because if the order is altered the result is that none of the Integrity Measurements comprising that PCR's Composite

[7] This topic is arguably disjoint as the concept of an Integrity Measurement Record is not introduced until 3.3.4. Its placement here, however, introduces the entire process in a single section.
[8] https://sourceforge.net/p/linux-ima/wiki/Home/

value can be checked therefore, they are not trusted. There is no known attack where changing the order of the Integrity Measurements will enable an attack on the integrity of the any individual Integrity Measurement.

A collection of Integrity Measurements is an Integrity Measurement Log (IML) or equivalently an Event Log.

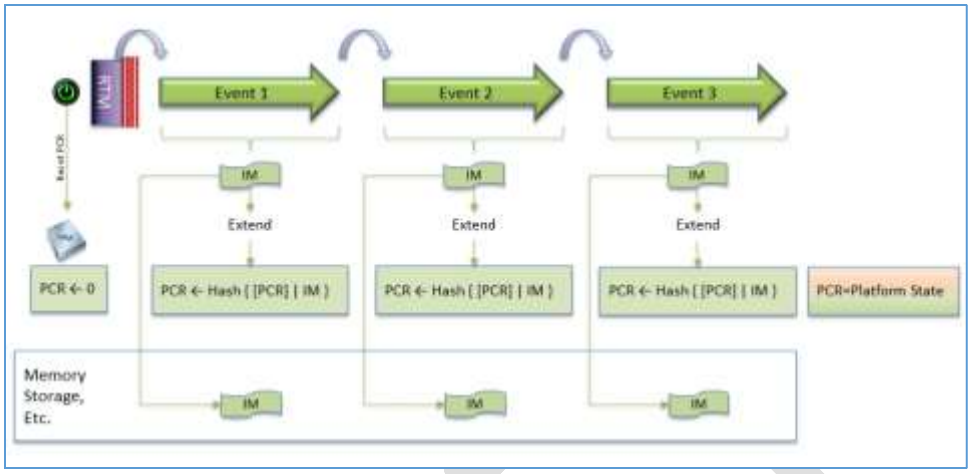


Figure 3 Measurement Process: Event Log (IML)

3.3.3 Event Log Metadata

A series of Integrity Measurements may be enhanced by wrapping additional information (metadata) about the Events into a structure. Examples of metadata are the PCR Index and a sequence number for each Integrity Measurement. While this metadata may be useful, and in some cases essential to the processing of the Event Log, the metadata is not a representation of the Event – only the Integrity Measurement is the representation of the Event. The following are examples of metadata that may be present in an Event Log:

1) Event Type

   a) As many types of Events may be included within a PCR's Composite value, providing the type of event associated with the Integrity Measurement may enable simpler lookup within a database of acceptable Integrity Measurement values.

   b) In the case where the order of the Integrity Measurements is irrelevant, the Event Type may assist in the alignment of the Attester's Event Log and the Expected Event Records.

   c) Assist the Verifier in cases where there may be different versions of the same Event (for example, an updated firmware) but the Verifier may not have information about all possibilities. For example, a system may be updated to a new firmware version but a known good value for that new firmware version is not yet known by the Verifier. Here an exception would be flagged but knowing the type of event may make isolating the problem and finding the "correct" value more efficient.

   d) Creating a RIM Event. See sections 5.2 RIM Provided by Reading Event Log in a Trusted Environment and 5.3 RIM Provided by Reading Event Log of a deployed device.

   e) Assisting developer and platform validation teams.

2) PCR Index

   While this guidance document uses a system with a single PCR for simplicity, platforms typically have multiple PCRs. On some systems the Integrity Measurement creation process may not occur in the same order as the PCR Indexes (I.e., there may be several measurements in PCR[0], followed by some into PCR[1], then more back into PCR[0].) Therefore, if the Integrity Measurements are stored in the sequence, they were made regardless of the PCR Index (the most efficient way for some environments such as pre-OS firmware) there needs to be a way to distinguish the Integrity Measurements made into each PCR. Also, measurement, storage, and transmission may be easier by avoiding segregating Event Log Records into different regions for each PCR.

3) Event Data

The content and purpose of this metadata is determined by the Event Type.

a) Event Data may provide additional details of the Event such as the version string for an Executable Event. While the Integrity Measurement value that is extended into the PCR is the full representation of the Executable Event, this information by be helpful when an unexpected Event is encountered.

b) Event Data may contain the actual Event if the Event is small. In this case, the information about the Event may be copied directly into the Event Data field or may be reformatted with the formatting information known to the Verifier. An example is the RTM code. Measurement of an RTM is not critical as it is a Root of Trust and therefore assumed to behave correctly. However, there may be value in measuring the RTM's version string. Another example is that small configuration strings may be placed in this Event Data field. In this example, the measuring component (which is already itself measured, therefore trusted) places relevant configuration information (for example the set of USB ports that are enabled as a bit map) into the Event Data area and the Event Data is measured.

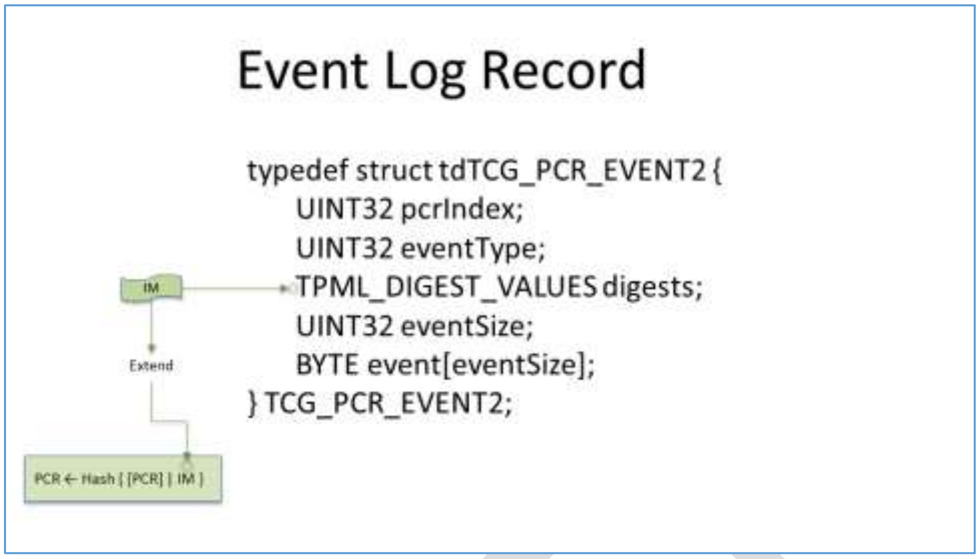
4) Event Sequence

The sequence of the Events (and therefore the Integrity Measurements) is critical but may be maintained in several ways. One method is by sequential memory location or strict positioning within a file. The PC Client's firmware is required to maintain Event Log sequence by strict sequential memory location. The firmware or OS may provide methods to read the Event Log directly into a file. Provided this method outputs the Event Log in the same sequence as they were in memory, the sequence is maintained in the file. However, some cases, for example, when the Event Log is transported, there may be a requirement to add an Event Sequence field to each Event within the Event Log.

3.3.4 PC Client Event Log Record

The PC Client Platform Firmware Profile [3] defines a profile of the TPM that is specific to the PC ecosystem. It includes an Event Log Record definition that is meaningful within the context of a PC ecosystem[9] as depicted in Figure 4. The PC Client Event Log Record called TCG_PCR_EVENT2 that contains a field named "digests" to hold the Integrity Measurement. The record also contains metadata such as "pcrIndex," "eventType," and an "event" field to assist the Verifier in parsing the records and managing exceptions. Figure 5 depicts the series of these Event Log Records as they are recorded or collected.

[9] This figure shows the Event Log structure for a TPM Family 2. In TPM Family 1.2 this structure is tdTCG_PCR_EVENT and this field is "UNIT8 digest[20]".



Event Log Record

```
typedef struct tdTCG_PCR_EVENT2 {
    UINT32 pcrIndex;
    UINT32 eventType;
    TPML_DIGEST_VALUES digests;
    UINT32 eventSize;
    BYTE event[eventSize];
} TCG_PCR_EVENT2;
```

Figure 4 Event Log Record

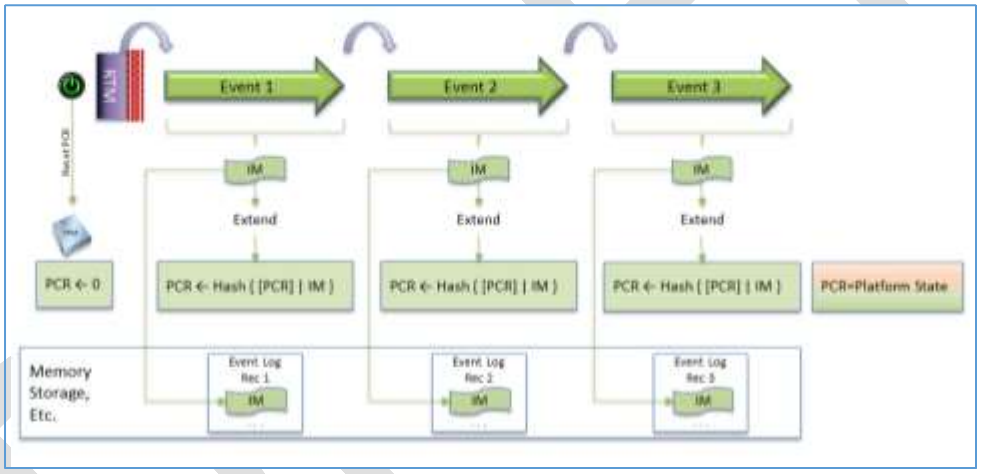


Figure 5 IM within Event Log

4 Verification

This section discusses the verification process for both PCR Composite and Event Log methods. PCR Composite is discussed first as it is the simpler of the two, as it potentially does not require sending the event log to the verifier.

It is important to note that the Verifier obtains information from the attester using one of TCG's Trusted Attestation Protocol (TAP) [4] protocols or others. The information must include a Quote and may optionally include event log values and an AK Certificate. The verification process requires multiple pieces of information from the Attester and the Asserter.

4.1 Getting a Quote

The Verifier must trust the authenticity of the source and integrity of the PCR values. However, these values are often transmitted via untrusted components (for example, drivers, network stack, applications) and media (for example, over a network). The Quote operation provides both these protections. For the purpose of this guidance document a "Quote" is defined as a digital signature on the PCR Value(s) or hash of the values, signed by a key the Verifier trusts which is resident with the same TPM as the PCR values. There are several specific methods of obtaining a Quote, including TPM_Quote (TPM 1.2 devices), TPM2_Quote (TPM 2 devices), and reading the PCRs under a signed audit session (TPM 2 devices). The differences between these methods must be understood by the Verifier.

In TPM 1.2 devices, the **TPM_Quote** operation returns a signed set of all selected PCR values. This is the simplest basis for attestation, since it returns all selected PCR values, signed by the TPM. Unfortunately, the TPM 1.2 devices only support the SHA-1 hash operation, which may be considered too cryptographically weak for modern use cases.

TPM 2 devices have a **TPM2_Quote** [2] operation which is hash agile, with potentially multiple banks of PCRs based on different hashes and hash sizes. These PCR values may be too large to fit in a signature, so the TPM2_Quote signs a hash of the list of all selected PCR values in all selected banks. This supports the use of strong hashes, but complicates the attestation verification, as the signature no longer contains the actual PCR values. While it is tempting just to read the PCRs (with a TPM2_PCR_Read command) and send their values along with the TPM2_Quote, this can cause problems since the PCR read may not be synchronized with the TPM2_Quote (asynchronous new extends may occur between the read and Quote operations). In the normal case, the TPM2_Quote is simply sent with the Event Log, as the event log can be used by the verifier to calculate what the PCR values should be, and this list can be hashed and compared to the signature. This is the most common method used, and it is **described in detail in the example in section 7.**

The **Audit Session** method for Quoting PCR values is to read the PCR values within a signed audit session (using the TPM2_GetSessionAuditDigest command). There are two significant advantages to this method: first the signature is done on the actual PCR values read, so you obtain these values without synchronization issues. Second, reading the PCR values also provides a signature across the pcrUpdateCounter that is returned by the TPM2_PCR_Read, and this is critical for verification in use cases in which the attester has undergone hibernation. The disadvantages of the Audit Session method include that it is complex, and it requires Endorsement Authorization, which may require changes to the typical TPM provisioning, and may not fit all use cases.

To understand the need for a verified pcrUpdateCounter, consider the following hibernation scenario:

A machine is booted and extends events into its PCRs, and its Event Log.

Before hibernation, an attestation (Quote and Event Log) is sent to the verifier. This gives a verified list of the PCRs and events.

This machine goes through a hibernation cycle. At this point, the PCRs are reset, but the log remains the same, since memory contents are saved on disk across the hibernation.

After the reboot, a new attestation (Quote and Event Log) is sent to the verifier. The Quote represents just the events since reboot, but the event log contains all events.

The verifier removes all those things from the event log that were previously in the log, as listed in the previous attestation, and the new Quote should match the new events in the log.

This seems simple. The problem is how do you know you have not missed malware that was run after the first Quote, but before the hibernation? The PCR values will be reset to zero, and the malware may be able to remove its entries from the log before the hibernation.

Knowing it went through hibernation helps but is not sufficient.

The answer is to use the pcrUpdateCounter, which persists across the hibernation, and tells you the total number of extension events that happened. The number of events in the current (full) log should match the pcrUpdateCounter.

The pcrUpdateCounter is ONLY given out by the TPM2_PCR_Read command, so we must do a signed audit session over that command, to provide proof to the verifier. We get the PCR values at the same time.

Verifying the audit session can be tricky. One approach is for the verifier to use a TPM emulator. The verifier resets the emulated TPM, extends the appropriate events into the emulator, does an audit around it, and sees if the audit content matches what came from the physical TPM.

4.2 Obtaining a RIM from the Asserter

The Verifier will need to obtain a signed RIM from the Asserter. Each RIM provides Manufacturer, Model, and version information that can be used for correlating PCR Composites and Event Log information. Section 5 provides insight into obtaining RIM information.

4.3 PCR Value Verification

This section describes the verification process for PCR composite values.

4.3.1 Obtaining a trusted PCR Value

In PCR composite verification, one method is for the verifier to have a fixed list of asserted PCR composite values, so that the attester sends just the Quote (with the hash of the PCR values) and does not have to send either the full PCR values or the event log. Alternately, if the PCR values are sufficiently static, the PCR values can be read and sent to the verifier and verified by the Quote. If fixed PCR composite values are not known to the verifier, and the PCRs are not static, Event Log verification as described in 4.3 should be used, as the event log will need to be included anyway.

The process of obtaining a PCR Value Quote is described in the following simplified description of operations:

1. Attester
    a. Attester's software performs a Quote operation (with a challenge from the verifier)
    b. Attester sends the Quote to the Verifier, optionally with the AK certificate as part of the TAP exchange.
2. Verifier
    a. Verifier performs signature verification of the Quote using the AK certificate provided in the TAP exchange or provided separately by the asserter. This verifies that the signature on the message is correct, and that it came from the attester's TPM.
    b. Verifier extracts the challenge from the message and compares it to the expected value for freshness.
    c. Verifier extracts the hash across the selected PCR(s) from the message.

At this point the Verifier can check this hash against allowed values.

4.3.2 PCR Value Verification

Figure 6 illustrates how to evaluate a platform using PCR Value. Starting from the verified PCR Value which represents the Attester's set of Integrity Measurements. The Verifier then[10]

1. Obtains expected PCR values from the Reference Integrity Measurements (RIMs) within the RIM DB.
2. An appraiser component within the Verifier hashes the selected expected PCR values and compares this value to that found in the Quote's message .

Note that a RIM may contain a set of Expected PCR Values. Some RIMs may contain multiple Expected PCR Values each paired with a PCR Index where all Expected PCR Values must match their respective PCR. The RIM may also contain OR operators where the RIM contains a set of acceptable PCR Values (for example, the platform may have multiple authorized versions of the firmware). If this set of allowed values is too large, the verifier will have difficulty trying all possible combinations, and the Event Log method would be preferred.

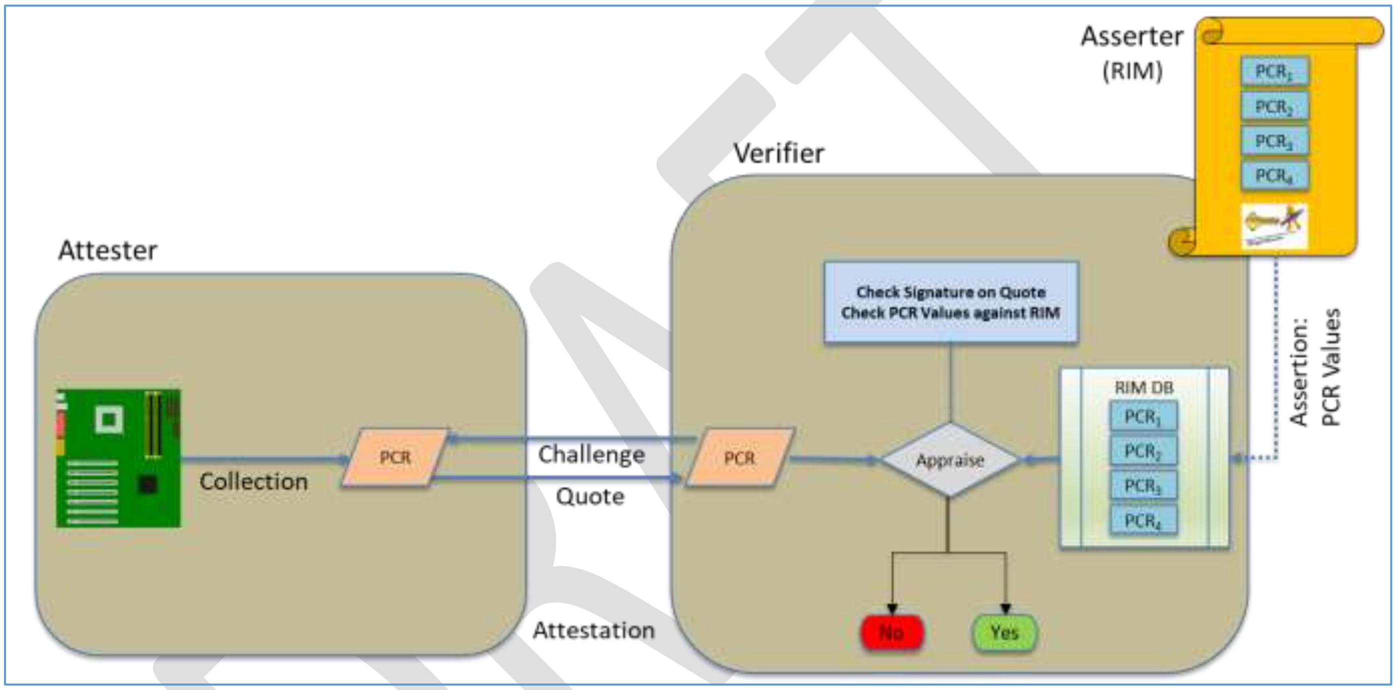


Figure 6 PCR Value Verification

4.3.3 PCR Value Benefits and Disadvantages

4.3.3.1 Advantages:

PCR Value Verification is simple to implement for uncomplicated or homogenous environments. It enables simple Verifier implementations and policies.

4.3.3.2 Disadvantages:

It is difficult to allow permitted but unordered Integrity Measurements within the Attester. PCR Value Verification will produce different PCR Values if measured Events are expected but are measured in different order. While coping with PCR Values in an arbitrary order is possible, any more than a few Events will create large permutations and a large and possibility complex RIM or set of RIMs.

The permitted situations where Integrity Measurements may be in unexpected order include (but are not limited to):

[10] The order described here (obtaining the PCR Value before obtaining the Expected values) is to retain the narrative flow. This order is not required in practice.

1. Platform Specs may be insufficiently prescriptive
2. Allowance for implementation variance
3. Multiprocessing (mostly during post-firmware (OS) measurements but some firmware implementations may allow multiprocessing.)
4. Lack of a “compliance” suite to drive consistent implementations
5. Add-in components may execute in different order depending on user

   For example, if a user plugin adapter contains a UEFI application or driver, Events will occur in different order depending on adapter’s slot
6. Relatively unrelated Events are mixed into same PCR
   - The PC Client Specification provides only eight pre-OS PCRs so there is some multiplexing

     For example, PCR [1] contains: BIOS setup parameters; CPU microcode patch; Platform configuration, etc.

4.4 Event Log (IML) Verification

4.4.1 Description

Figure 7 illustrates an Event Log checked against a Quote via “PCR Replay” technique in which the expected PCR values are calculated from the Event Log and these calculated values compared against the provided Quote. The digests within the event log are then checked against the RIM DB. The RIM DB contains reference measurements processed from Trusted RIM bundles

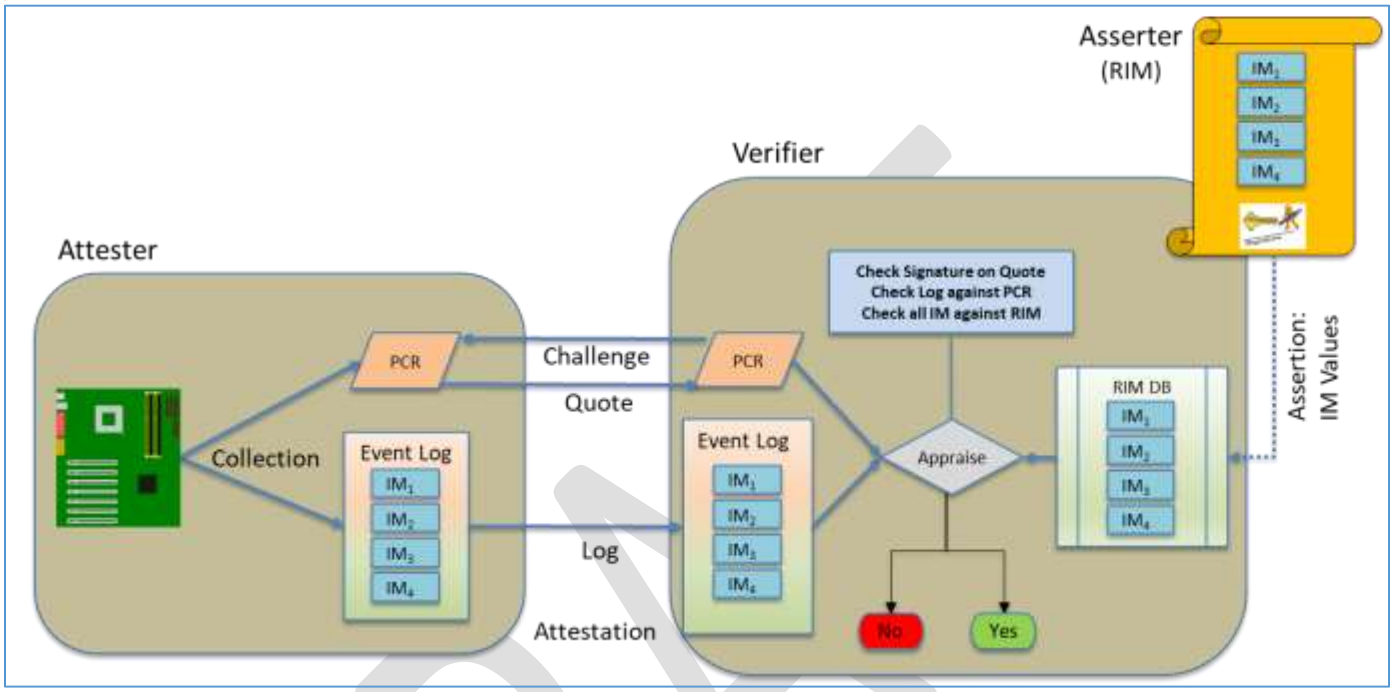


Figure 7 Event Log IML verification

4.4.2 Appraiser

The Appraise component decides the Attester’s overall trust state. The Appraiser must:

1. Retrieve the corresponding RIM for the Attester. Note that this may be done asynchronously (for example pushed by the Asserter) or sent synchronously by the attester as part of each attestation.
2. Obtain and verify a fresh Quote from the attester, which signs the current PCR values and challenge.
3. Obtain and parse the event log into discrete events and verify that each entry is correctly formatted.
4. Verify the integrity of the overall log by comparing the hash of the selected PCR(s) in the Quote to those calculated from the individual event digests via the PCR Replay technique.
5. Verify each Log entry against RIM data or against the expected event digest and data with local policy.

   NOTE: Not all Log Entries may be applicable to a Local Policy, therefore some may be skipped. However, is it critical that the entire Even Log be verified (as done in 4 above) before an any specific Log Entry can be trusted.

In the normal Event Log Verification, the attester sends just the Quote and the Event Log to the verifier, as the log is sufficient to calculate the claimed PCR values, and the Quote can verify these claims. There is normally no need to send the PCR values, and this simplifies the attester, as there is no race condition between the Quote and reading the PCRs. If the Event Log is not verified by the Quote, then sending the PCR values separately may help in debugging the Event Log itself, so long as the PCR values are verified by the Quote. (Since the Quote contains only a hash across all selected PCR values, it cannot tell which PCR(s) are incorrectly represented in the log, while a separate list of verified PCR values can help locate problems.)

A detailed example is given in Section 7. This example will cover the normal Event Log verification in which only the Quote and Event Log are sent.

A RIM for Event Log Verification is like the RIM provided for the PCR Value verification, except instead of providing a set of Expected PCR Values it contains a set of Expected Integrity Measurements. Policies (which may be provided by the Appraiser or may be within the RIM itself) may allow or prohibit variances such as Event ordering, allowance, or disallowance of additional Events, etc.

Metadata may be provided in the RIM to refine the Verifier's policies. As an example, if Event Type is provided by the RIM, the Verifier may also compare that against the Event Record's Event Type. While Event Type is not a representation of the Event, if provided by the RIM, it may be used to detect tampering of the Event Log in storage or transmission.

Table 2 is a set of example policies:

Index	POLICY DESCRIPTION	ATTESTER'S EVIDENCE	APPRAISER ACTION
1	Allowed List contains only the set of Event Log Records permitted in any order	Contains exact set	Allowed
		Contains entire set + 1	Disallowed
2	Allowed List contains only the set of Event Log Records permitted in exact order	Contains exact set in specified order	Allowed
		Contains exact set, but in different order	Disallowed

Table 2 Example Policies

5 Methods for obtaining RIMs

Figure 8 depicts three methods for creating a Reference Integrity Measurements (RIMs) for either PCR Values or Event Logs.

5.1 RIM Provided by the Event's Provider or Asserter

It is critical to the operation of a Verifier that the certificate paths used to verify the digital signature of a RIM are checked during each verification. RFC 5280 defines a standardized path validation algorithm for X.509 certificates, given a certificate path. One Certificate path is required from each of the event providers or asserters of a given device.

It is also critical that the Verifier is configured to use certificate paths that represent only the event providers that are trusted by the organization responsible for the security of the device (Not necessarily the default paths that delivered with a Verifier). The Root CA certificate(s) or a separate fingerprint of the Root CA certificate (e.g., SHA256 hash of the certificate) should be delivered out of band (not on the device) and the Root CA should be trusted using standard methods outside the scope of this document..

It Is further suggested that each organization making use of TCG Verification add a deliverable requirement for RIM and associated certificate paths on all procurement requirement documents for all device that require Integrity Measurement Verification. Even though a device may contain a TPM the device manufacturer is not obligated to provide RIMs for a given device delivered by default.

5.2 RIM Provided by Reading Event Log in a Trusted Environment

If the OEM does not produce RIMs, they may be created by a trusted entity (for example a VAR or a company's IT department while evaluating a platform). This is like the example provided in section above, where the OEM reads the PCR Values or Event Log. The only difference is that the RIM provider must have confidence in the supply chain between the OEM and the RIM provider.

5.3 RIM Provided by Reading Event Log of a deployed device

This is like the section above, except the PCR Values or Event Logs may be obtained, as shown in Figure 8, from deployed platforms using software on an Attester's platform in a production environment. This is less trusted than any method discussed in this section as the platform may have already been compromised. Therefore, the Event Records including the Integrity Measurements and even the PCR Value may not be valid.

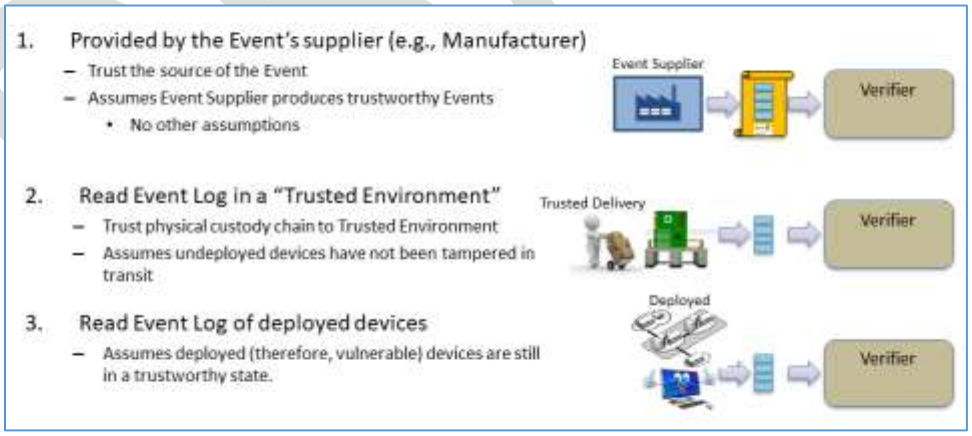

1. Provided by the Event's supplier (e.g., Manufacturer)
– Trust the source of the Event
– Assumes Event Supplier produces trustworthy Events
• No other assumptions
Event Supplier
Verifier
2. Read Event Log in a "Trusted Environment"
– Trust physical custody chain to Trusted Environment
– Assumes undeployed devices have not been tampered in transit
Trusted Delivery
Verifier
3. Read Event Log of deployed devices
– Assumes deployed (therefore, vulnerable) devices are still in a trustworthy state.
Deployed
Verifier

Figure 8 Producing Assertions (Reference Measurements)

6 Timeline

Figure 9 illustrates a timeline example of actors and processes described in this guidance.

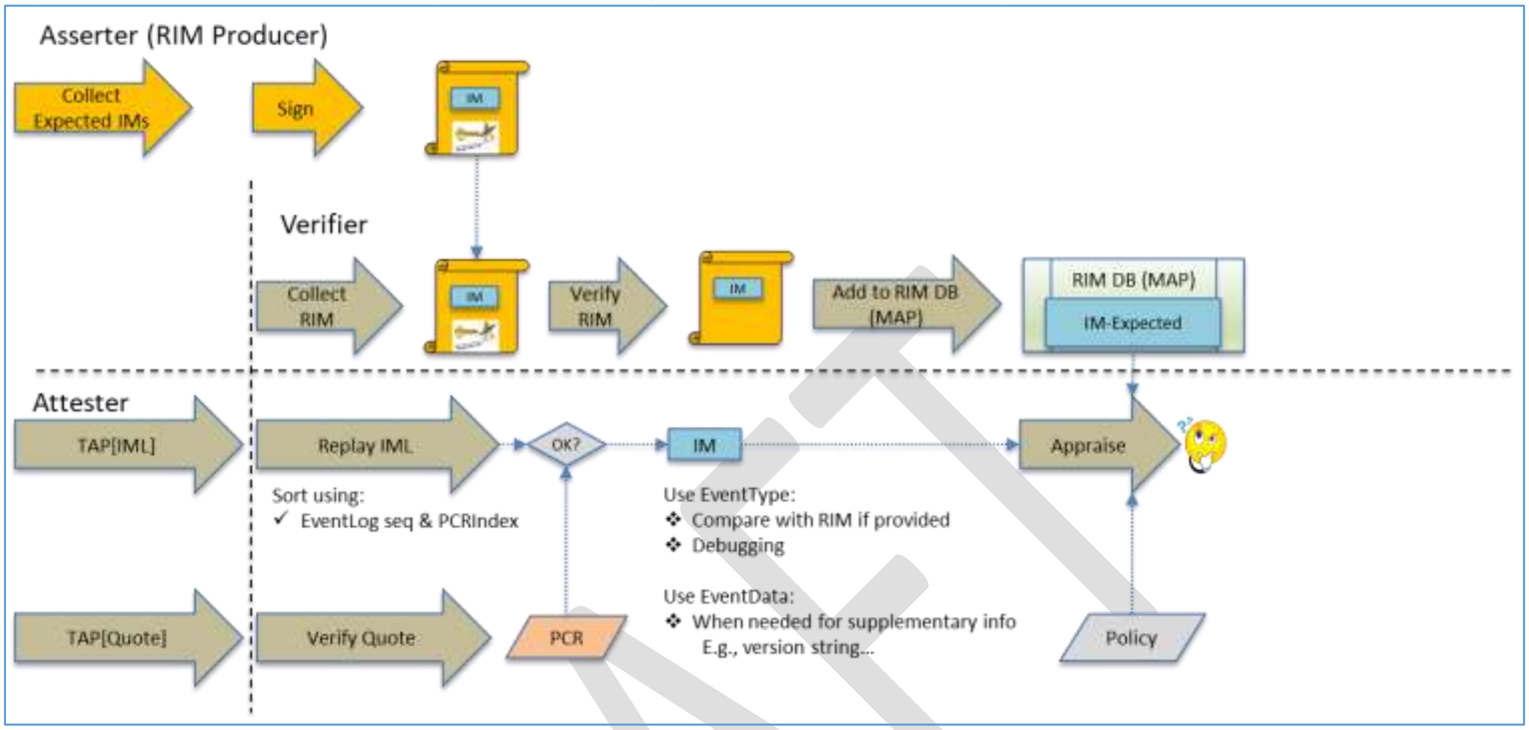


Figure 9 Example Verification Timeline

Starting at the top left, section 5 describes how Reference Integrity Measurements (RIM) are placed into an Event Log. RIMs are provided to a Verifier. In the lower left, the Attester creates Integrity Measurements as described in section 3. The RIM(s) and the Event Log is sent to the Verifier. Sending the Event Log is called Attestation. Section 4 describes how the Verifier parses the Event Log and the RIMs and using a policy arrives at a trust decision.

7 An Example: PCClient Event Verification for a Linux Endpoint

The previous sections provide an abstract description of Collection and Verification of integrity logs in general. This section provides a detailed look at collection and verification on an example UEFI based PC running Fedora Linux, with secureboot turned on, with a Microsoft signed shim, with GRUB and the kernel signed by Fedora. It provides guidance on how to interpret and use the respective TCG standards to implement verification safely. This example will cover only the UEFI collected PCClient events. It will not cover subsequent GRUB and Linux generated events for simplicity. It will show examples of how the event logs and attestation data are collected from the Linux system after boot.

7.1 PCClient Boot-time Integrity Collection on a Linux System

Before we can look at verification of the log entries, we must first understand exactly what is collected, and how it is collected.

UEFI systems provide a HashLogExtendEvent service for simplified TPM based event logging during the boot process. It is shown in Figure 10 and described in detail in the TCG EFI Protocol Specification [7]. This logging service is used not only by the UEFI (BIOS) itself but also by the various bootloader phases such as Shim and GRUB prior to the operating system being able to make its own log. The TCG PC Client Platform Firmware Profile Specification [3] details the required BIOS event logging, while the bootloader and operating systems log entries are not standardized by TCG and will not be covered in this example.

The HashLogExtendEvent service takes three main arguments:

DataToHash	address of buffer to be hashed
DataToHashLen	length of data to be hashed
EfiTcgEvent	Pointer to information about the Event, including eventType, pcrindex, and “event” data

This service hashes the provided buffer and extends the indicated PCR with the result. The digest and event fields are then added to the event log. The results are logged in a TCG_PCR_EVENT2 structure as shown in the figure 10. Note that the green fields in the logged TCG_PCR_EVENT2 structure are fully verifiable by the TPM, while the red fields are not, as they are not included in the digest sent to the TPM.

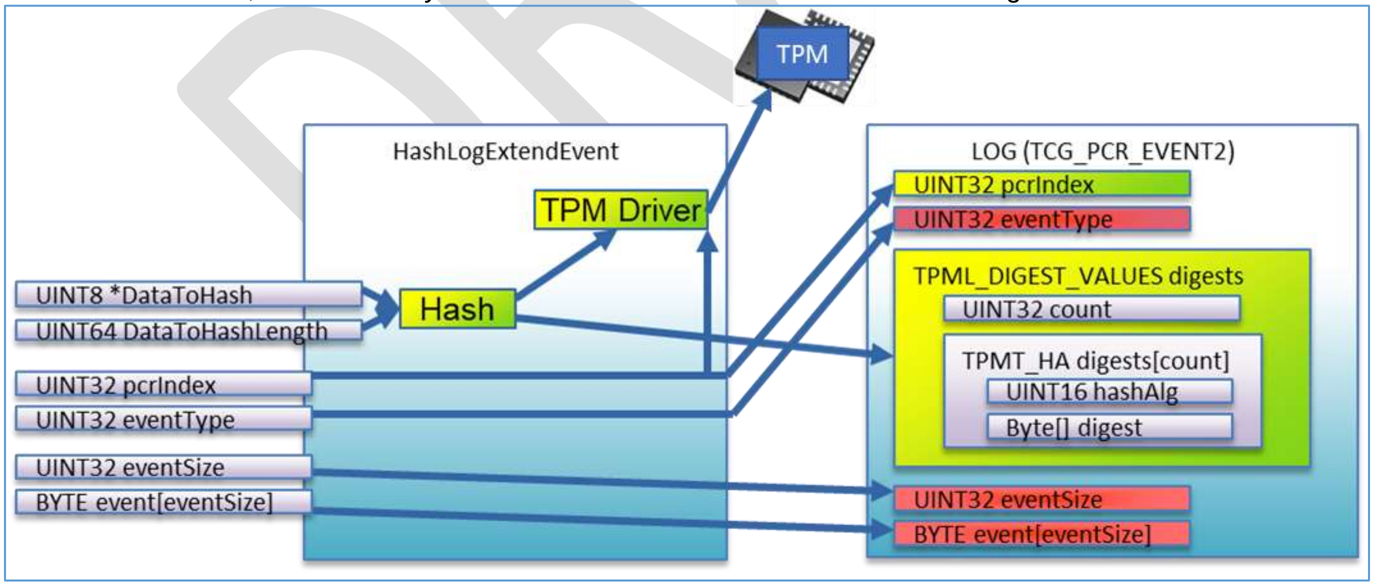


Figure 10. HashLogExtendEvent processing

HashLogExtendEvent was designed to be simple for the callers to use, and to support multiple use cases, of which remote attestation is just one. It wanted to hide the details of cryptography, the TPM, and the Event log handling from the callers. It also wanted not to cause longer boot times, so emphasis was placed on re-using data and hashes already calculated by UEFI as part of Secure Boot. But since only the digests contribute to the PCRs which are signed by the TPM, the eventType and Event data may be completely unverified. The details of exactly what is contained in the eventType and Event data fields is specified in [3]. In some cases, the data is the same as the DataToHash buffer, so the contents are fully verifiable. In some cases, the Event data field is not the same as the hashed buffer, but it contains data that is not strictly needed for verification. In all cases, the verifier must be careful with the eventType and Event data fields, since even if all digests in the event log match the Quoted PCR values, this does not necessarily validate the claimed eventType or Event data fields.

The UEFI collected log is stored in the ACPI system, where the operating system can read it after boot. The Linux kernel makes the raw (binary) log available for reading as a pseudo-file at:

```
/sys/kernel/security/tpm0/binary_bios_measurements
```

7.2 PCClient Boot-time Integrity Log Analysis for a Linux System

Given the event log as a binary blob, how do we verify it? Section 4 listed seven steps for event log verification. Let us look at these steps in detail for the example system.

7.2.1 Obtain a RIM from a PC Client Attester

Before a Verifier can perform a verification, it needs reference measurements. There are several methods that can be used to retrieve a RIM Bundle for the Attester:

1. The Event log contains a tdTCG_Sp800_155_PlatformId_Event that contains a ReferenceManifestGuid which can be used by the Verifier to match an event log to a specific RIM Bundle's tagId attribute.
2. For PC Client system a RIM Bundle can be placed by the OEM in the EFS partition when the device is created. The PC Client RIM Bundle includes a Support RIM file that follows the Event Log format and a signed Base RIM that provides integrity protection of the Event log via a hash within its payload. The base RIM includes a pcUriGlobal attribute in which provide a location from which supplemental and patch RIM Bundles can be obtained.
3. If the device contains a TCG Platform Attribute Certificate then the verifier can use the Manufacturer Name, Model, and version within the certificate to match the attester to a specific a RIM Bundle. The Platform Attribute Certificate contains a PlatformConfigUri attribute which provide a location of where to find a RIM Bundle for the specific type of Attester.

The RIM Bundle can be stored by the verifier or processed and placed in the RIM Database (DB). The RIM DB should preserve any Meta data needed to recall RIM data from the attester including event digest values and any Meta Data needed for the event log verification.

For this example, we assume that all necessary data is already in the appraiser's RIM DB. In the following sections we will highlight exactly when and what RIM data is needed for the example system. As will be shown, only a small amount of RIM data is needed. The minimal needed for the example system is trusted EFI hashes for the bootloader (shim.efi). All other events can be verified based on the TPM's PCR values.

7.2.2 Obtain and verify a fresh TPM2_Quote from the Attester

For Linux, obtaining and verifying a TPM2_Quote is most easily understood at first by experimenting with the tpm2_quote and tpm2_checkquote command line tools in the tpm2-tools package.

A simple example is:

On the attester:

```
#tpm2_createprimary -C e -c primary.ctx
#tpm2_create -C primary.ctx -u key.pub -r key.priv
#tpm2_load -C primary.ctx -u key.pub -r key.priv -c key.ctx
#tpm2_quote -Q -c key.ctx -l 0x0004:10 -m msg.bin -s sig.bin -q deadbeef
```

Here key.pub is the public key which will be used by the verifier to check the signature. "msg.bin" is the message containing the challenge and hash across the selected PCRS (in this case, just the sha-1 bank of PCR 10). "sig.bin" is the TPM's signature on msg.bin. "deadbeef" is the hexadecimal freshness challenge from the verifier.

On the verifier:

```
tpm2_checkquote -u key.pub -m msg.bin -s sig.bin -q deadbeef
```

At this point, the verifier knows that msg.bin and sig.bin came from the correct TPM and are untampered. The verifier should next check that msg.bin contains the expected challenge. Also, the verifier should extract the hash of the selected PCR(s). In this example, the sha-1 value in PCR-10 was

```
    EA3A8AB9A29C7AF24A492CAFB8FD52A85DC6B370
```

and the expected sha256 hash of this should be

```
    e0a0879966d6498ed6937ca5860f0abe872d14fd8a010a57d8eefee054703a33
```

The following is a hex dump of msg.bin, showing the challenge in yellow and the hash in green:

```
[root@localhost ~]# hexdump -C msg.bin

00000000  ff 54 43 47 80 18 00 22  00 0b a2 30 b0 19 e0 df  |.TCG..."...0....|
00000010  65 f4 94 12 95 ed 37 a2  20 66 ae 36 5d 39 37 c9  |e.....7. f.6]97.|
00000020  7e fe 82 32 e6 cf c8 07  47 10 00 04 de ad be ef  |~..2....G.......|
00000030  00 00 00 00 13 7c 0a 56  00 00 00 50 00 00 00 00  |.....|.V...P....|
00000040  01 00 03 00 25 00 00 00  05 00 00 00 01 00 04 03  |....%...........|
00000050  00 04 00 00 20 e0 a0 87  99 66 d6 49 8e d6 93 7c  |.... ....f.I...||
00000060  a5 86 0f 0a be 87 2d 14  fd 8a 01 0a 57 d8 ee fe  |......-.....W...|
00000070  e0 54 70 3a 33
```

Now the verifier knows that this Quote is fresh, and it knows the hash of the Quoted PCR(s).

Figure 11 shows the virtual data flow for this simple case:

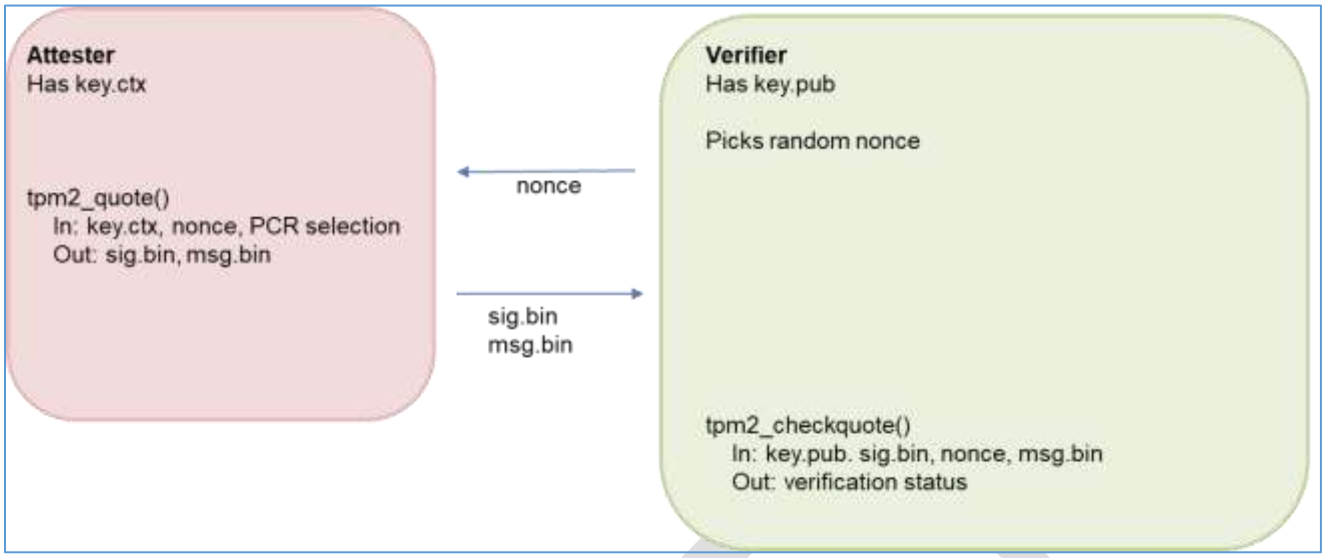


Figure 11: Example Attestation Quote Data Flow

First the attester creates an attestation key ("key") and sends the public part (key.pub) to the verifier. For each attestation, the verifier picks a random nonce. The attester uses tpm2_quote, which outputs the signature (sig.bin), and the overall message that was signed (msg.bin). These are sent to the verifier, which uses tpm2_checkquote to verify the Quote with its trusted copy of the public key and the nonce.

Assuming the verification was successful, the verifier knows that the pcrs.bin contains a hash of the selected PCR value(s), against which the event log can be verified.

For production use, the TCG Trusted Attestation Protocol (TAP) defines a set of information elements sent between the attester and the Verifier, and their standardized encodings. The developer can operate the protocol based on the client pushing or the verifier pulling the attestation, or a combination. Several reference implementations for Attestation, including making and verifying Quotes exist, including HIRS [8], and IBMACS [9]

Synchronization Issues

In this example case we are looking just at events prior to the OS. Once the OS is running, these events are stable, and synchronization between the TPM2_Quote and reading the PCClient log is not required, If OS level events are also to be verified, however, it is critical to obtain the TPM2_Quote first, before reading the measurement log. On Linux, a TPM2_Quote operation is a separate, asynchronous operation from reading the measurement log. If the TPM2_Quote is done first, then we know that the measurement list must contain the same or more events, but not less. The verifier can check after each event in the log to see if there is a match with the Quote. There could be extra events at the end of the log, but at some point there must be a match to the Quote. If there are extra unverified events at the end of the list, the verifier should simply ignore them as they cannot be verified. (On a running system, and attestation is always a "point-in-time" verification, which should be understood as occurring at the time of the Quote). If, on the other hand, we were to read the list first, and then get the Quote, the log may be missing the latest events, and there would be no way to verify the list against the Quote, and a new attestation would be required.

7.2.3 Obtain and parse the log into discrete events

The event log found in Linux at /sys/kernel/security/tpm0/binary_bios_measurements is a single binary stream of data as shown in Figure 12. This stream of data may optionally be translated by the attestation application into the Canonical Event Log Format (CEL) as specified in [11]. For simplicity, this guidance document describes only the original native format. The CEL specification gives detailed examples of logs in native and translated formats with detailed parsing information. The native log must first be parsed into individual events

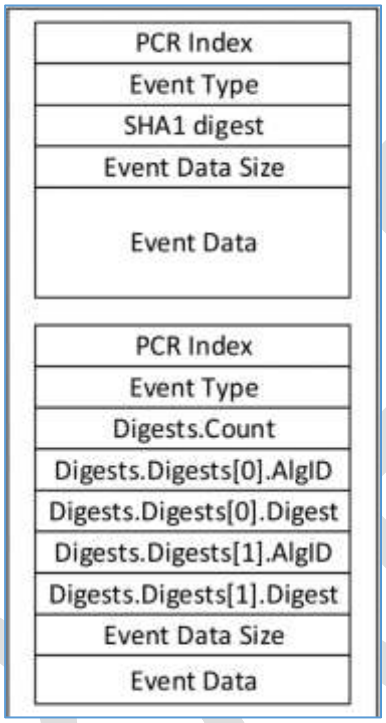


Figure 12: Native Event Log Format

The first event is always logged with a fixed SHA1 digest field, for backwards compatibility. All subsequent events are logged in the hash agile format that supports multiple hashes of different types. The Event Data and Digest fields are byte arrays of varying lengths. All other fields are UINT32 in Little-Endian format.

As the event log is unverified at this stage, great care must be taken to avoid data driven attacks, particularly for the variable length event data, and the variable count digest fields.

.

7.2.4 Verify the integrity of the overall log by comparing to actual PCR values

Once the individual events are parsed, virtual PCR values can be calculated by extending the event digests for the indicated PCRs for all events (PCR Replay). If the calculated PCR values match the hashed PCR values from the Quote, then we know that the digests are untampered. If any of the PCR values do not match, then the entire log is suspect.

Calculating the PCR values requires simulating the extend operation that the TPM performs. The pseudo-code for this is:

Initialization: PCR = 0;

For each extend operation: PCR = hash(PCR | digest)

That is, all PCRs start out with a value of zeros, and each new value is the result of the hash of the old value concatenated with the new event digest. As discussed in 7.2.2, this verification should in general check for matches to the Quote after each event, to handle the case of events that occur after the Quote. In this particular example we are verifying only past firmware events, so there should be no events after the Quote, but checking after each event is a good habit which will handle the general case.

7.2.5 Verify each Log entry

7.2.5.1 Verify the integrity and correctness of select log entries

Once the event digests have been verified against the Quote we know that the event digests, PCR indices, and order are untampered. The next step is to examine each event, and attempt to verify through the digest, either because the digest is for a binary which is in the RIM, or because the digest matches the event data directly.

First, it is important to remember that the eventType field is not verified and may have been altered by an attacker. The verifier can use it only as a hint for verifying the event data. For example, if the eventType is EV_EFI_BOOT_SERVICES_APPLICATION, then the verifier would attempt to find the digest in the RIM list of EFI hashes. Since the digests have been verified, finding the digest in the RIM's EFI hashes would strongly verify that the eventType was correct.

it is also critical to understand the details of how the data was collected, as discussed in section 7.1. As some of the fields are not validated by the digest, depending on the use of HashLogExtendEvent, we may have to verify the integrity and correctness of the remaining event fields in some other way. For the example system, the event log entries can be verified in one of five ways:

CONTENT_MATCHES_DIGEST

In most of the cases, the event content was small, and was the same data that was hashed, providing full cryptographic protection of the event data content. This protection was not perfect: in these events, the event type tags were still not protected as was just discussed. Fortunately, the digests and event data were sufficient in all cases to provide safe verification.

HASH_MATCHES_EFI

In these events, a large binary is being hashed, and the event content has unauthenticated "hints" about the binary, such as an EFI path to the file being loaded. While the attacker could freely modify the claimed path, the hashes are authenticated by a RIM DB of approved EFI binaries, and the event content are then verified to match to the "approved" path. An attacker could only force the content not to match. In the example log, the RIM must provide an EFI hash for shim.efi. Note that these hashes must be the "normalized" EFI hashes, which are not a simple hash of the file, but are hashes of the file excepting the space for the EFI signature(s). The source code for utility sbverify in the sbsigntools repository [10] can be used as an example of how to calculate the normalized EFI hash of a file.

AUTHORITY_FOUND

In this case, the event content is the actual binary certificate data that was used to verify the EFI image being loaded. The event content is both matched to the digest, and directly to the EFI variable containing the "approved" certificate data. A verifier could stop here, but it could even take the further step of validating the certificate chain itself. As mentioned earlier, this strongly authenticated certificate cannot be used to verify the loaded binary directly, as the signature itself is not logged, so the verifier still needs to find the binary's hash in the trusted RIM database.

NONE

The first event (EV_NO_ACTION) is by specification completely unauthenticated – no PCR is extended, so no digest can be verified, and the content is not hashed anyway. This content can be changed arbitrarily by the attacker, and a verifier must simply ignore its content. Table 3 summarizes the event verification methods used in the example log in Appendix A.

Event Type	Count	Verification	Notes
EV_NO_ACTION	1	NONE	No digest extended, => must ignore

			content
EV_S_CRTM_VERSION	1	CONTENT_MATCHES_ DIGEST	(TYPE not authenticated)
EV_EFI_VARIABLE_DRIVER_CONFIG	5	CONTENT_MATCHES_ DIGEST	(TYPE not authenticated)
EV_POST_CODE	1	HASH_MATCHES_EFI	Hash in RIM
EV_SEPARATOR	8	CONTENT_MATCHES_ DIGEST	(TYPE not authenticated)
EV_EFI_GPT_EVENT	1	CONTENT_MATCHES_ DIGEST	(TYPE not authenticated)
EV_EFI_VARIABLE_BOOT	6	CONTENT_MATCHES_ DIGEST	(TYPE not authenticated)
EV_EFI_BOOT_SERVICES_APPLICATION	2	HASH_MATCHES_EFI	Get “normalized” hashes from RMI (Path not directly validated)
EV_EFI_VARIABLE_AUTHORITY	1	AUTHORITY_FOUND	Match content to specified efivar. This also means that corresponding EFI binary load was authenticated with this cert.
	26		

Table 3: Log Validation Summary

In summary, the event log in Appendix A shows twenty-six events. So long as the RIM data includes the hashes for the three HASH_MATCHES_EFI events (such as `/boot/efi/EFI/fedora/shim.efi`), then all events can be fully validated, with the understanding that the given EFI paths should not be trusted.

7.2.5.2 Verify that the overall log is correct (compliant with TCG content and ordering standards).

At this stage we know that the list is untampered, and each event is correct. The next step is to verify that the overall log contains all entries necessary to judge the attester’s security. One way to do this is to check that the log is compliant with the respective standards, such that it includes all the mandatory (necessary) entries, and that all the entries are in the correct order. The omission or mis-ordering of events could open vulnerabilities in the measured boot sequence, which would otherwise not be noticed simply by verifying each individual event.

The example log in Appendix A has mis-ordered events. (Some of the SEPARATOR events occur too early in the log.)

Verifiers should be careful to note any such problems with the overall list.

While the detailed event log requirements are specified in [3], here are examples of some of these requirements.

PCClient Required Events: the following events must be included:

EV_NO_ACTION

EV_S_CRTM_VERSION

EV_EFI_VARIABLE_DRIVER_CONFIG (must log PK, KEK, DB, and DBX variables)

EV_POST_CODE

EV_EFI_GPT_EVENT

EV_EFI_VARIABLE_BOOT (must log BootOrder and Boot#### variables)

EV_SEPARATOR

EV_EFI_VARIABLE_AUTHORITY

EV_EFI_BOOT_SERVICES_APPLICATION

PCClient Reserved or Deprecated Events (should not be included):

EV_PREBOOT_CERT

EV_UNUSED

EV_IPL

EV_IPL_PARTITION_DATA

PCClient Required ordering:

EV_NO_ACTION must be the first event

EV_SEPARATOR for a given PCR must come after all events for that PCR

and before the first EFI application

7.2.5.3 Verify that the overall log meets policy.

Finally, local policies can be enforced on the overall log. While the standards specify event types and some ordering, they do not cover the operating system requirements or other semantics in the optional events. For example, the log in Appendix A implies that a boot of Fedora 32 should include the following boot sequence:

(CRTM) -> EFI Config -> EFI Boot Variable -> shim.efi

A local policy may require that all these elements be contained in the log in this order, which goes beyond the basic requirements in the TCG specifications.

Again, this verification may require obtaining a known good reference log against which to compare.

7.2.6 Summary of the Event Log's Verification

The listing in Appendix A is the verifier output for the example system, where the verifier has followed the previous steps in 7.2. Given the understanding of the integrity collection phase, it is critical for the verifier to understand which parts of a given log are verifiable by the Quote, and which parts are not. In this listing, the yellow highlight indicates parts of the log which are not verified by the Quote. The verifier must not trust or rely on any of the yellow parts in its analysis. The green color highlights entries that are strongly verified with digests that directly match RIM data, such as for the loaded EFI binaries and trusted authority certificates.

For simplicity, this listing shows the basic summary which for each record includes the PCR number, the eventType, and a summary of the event data and verification result. The twenty-six events are from the BIOS and are compliant with the TCG specification.

Since the calculated PCR values for this event log match the actual PCR values, we know that sequence numbers, PCR numbers, and digests have not been tampered. The green highlights show event data directly validated from digests matching RIM data. The yellow highlights are of the unverifiable eventType and hints. Despite the limitations of HashLogExtendEvent, all events other than the initial EV_NO_ACTION event are verifiable, given trusted RIM hash/path data. The table in 7.2.5 summarized the event types and verification for all events in that log, and the paragraphs explained the different types of validation results listed in the table.

Most of the yellow fields within events are of no real concern – while the paths are not verified, the binaries are authenticated. The critical events are the ones shown in green, which verify the loaded EFI binaries. This also includes the respective AUTHORITY events which provide the certs used in the secure boot.

8 References

[1] "TCG Trusted Network Connect TNC Architecture for Interoperability"; Specification Version 1.1 Revision 2 1 May 2006, https://trustedcomputinggroup.org/wp-content/uploads/TNC_Architecture_v1_1_r2.pdf

[2] "TPM 2.0 Library Specification"; Family "2.0" Level 00 Revision 01.38 September 29, 2016, https://trustedcomputinggroup.org/resource/tpm-library-specification/

[3] "TCG PC Client Platform Firmware Profile Specification"; Family "2.0" Level 00 Revision 1.04 June 3, 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientSpecPlat_TPM_2p0_1p04_pub.pdf

[4] "TCG Trusted Attestation Protocol (TAP) Information Model", Version 1.0, Revision 0.29A,1/11/2019, https://trustedcomputinggroup.org/wp-content/uploads/TNC_TAP_Information_Model_v1.00_r0.29A_publicreview.pdf

[5] "TCG PC Client Platform Firmware Integrity Measurement Specification"; Version 0.15March 31, 2020, https://trustedcomputinggroup.org/wp-content/uploads/TCG_PC_Client_RIM_r0p15_15june2020.pdf

[6] "TCG Reference Integrity Manifest (RIM) Information Model (IM)"; Version 1.00, Revision 0.13December 4, 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG_RIM_Model_v1-r13_2feb20.pdf

[7] "TCG EFI Protocol Specification", Family "2.0" Level 00 Revision 00.13 March 30, 2016, https://trustedcomputinggroup.org/wp-content/uploads/EFI-Protocol-Specification-rev13-160330final.pdf

[8] Host Integrity at Runtime and Start-up (HIRS) https://github.com/nsacyber/HIRS

[9] IBM TPM Attestation Client Server, https://sourceforge.net/projects/ibmtpm20acs/

[10] Sbsigntools https://git.kernel.org/pub/scm/linux/kernel/git/jejb/sbsigntools.git/tree/docs

[11] "TCG Canonical Event Log Format," https://trustedcomputinggroup.org/resource/canonical-event-log

Appendix A: An Analyzed PCClient Event Log

```
PCClient: BIOS Events
Legend: Verified by Digest, Verified by RIM, Unverified

SEQ 0 PCR 0 DIGESTS SHA1 PCCLIENT EV_NO_ACTION
SEQ 1 PCR 0 DIGESTS SHA1 SHA256 PCCLIENT EV_S_CRTM_VERSION CONTENT_MATCHES_DIGEST
SEQ 0 PCR 7 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_DRIVER_CONFIG SecureBoot CONTENT_MATCHES_DIGEST
SEQ 1 PCR 7 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_DRIVER_CONFIG PK CONTENT_MATCHES_DIGEST
SEQ 2 PCR 7 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_DRIVER_CONFIG KEK CONTENT_MATCHES_DIGEST
SEQ 3 PCR 7 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_DRIVER_CONFIG db CONTENT_MATCHES_DIGEST
SEQ 4 PCR 7 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_DRIVER_CONFIG dbx CONTENT_MATCHES_DIGEST
SEQ 5 PCR 7 DIGESTS SHA1 SHA256 PCCLIENT EV_SEPARATOR CONTENT_MATCHES_DIGEST
SEQ 2 PCR 0 DIGESTS SHA1 SHA256 PCCLIENT EV_POST_CODE
SEQ 0 PCR 4 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_BOOT_SERVICES_APPLICATION SHA256_MATCHES_EFI BIOS
            CONTENT_NO_MATCH
SEQ 3 PCR 0 DIGESTS SHA1 SHA256 PCCLIENT EV_SEPARATOR CONTENT_MATCHES_DIGEST
SEQ 0 PCR 1 DIGESTS SHA1 SHA256 PCCLIENT EV_SEPARATOR CONTENT_MATCHES_DIGEST
SEQ 0 PCR 2 DIGESTS SHA1 SHA256 PCCLIENT EV_SEPARATOR CONTENT_MATCHES_DIGEST
SEQ 0 PCR 3 DIGESTS SHA1 SHA256 PCCLIENT EV_SEPARATOR CONTENT_MATCHES_DIGEST
SEQ 1 PCR 4 DIGESTS SHA1 SHA256 PCCLIENT EV_SEPARATOR CONTENT_MATCHES_DIGEST
SEQ 0 PCR 5 DIGESTS SHA1 SHA256 PCCLIENT EV_SEPARATOR CONTENT_MATCHES_DIGEST
SEQ 0 PCR 6 DIGESTS SHA1 SHA256 PCCLIENT EV_SEPARATOR CONTENT_MATCHES_DIGEST
SEQ 1 PCR 5 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_GPT_EVENT CONTENT_MATCHES_DIGEST
SEQ 1 PCR 1 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_BOOT CONTENT_MATCHES_DIGEST
SEQ 2 PCR 1 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_BOOT CONTENT_MATCHES_DIGEST
SEQ 3 PCR 1 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_BOOT CONTENT_MATCHES_DIGEST
SEQ 4 PCR 1 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_BOOT CONTENT_MATCHES_DIGEST
SEQ 5 PCR 1 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_BOOT CONTENT_MATCHES_DIGEST
SEQ 6 PCR 1 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_BOOT CONTENT_MATCHES_DIGEST
SEQ 6 PCR 7 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_VARIABLE_AUTHORITY AUTHORITY_FOUND CONTENT_MATCHES_DIGEST
SEQ 2 PCR 4 DIGESTS SHA1 SHA256 PCCLIENT EV_EFI_BOOT_SERVICES_APPLICATION SHA256_MATCHES_EFI
            /boot/efi/EFI/fedora/shim.efi CONTENT_NO_MATCH


Sha1 predicted and actual:
PCR 0: FCCCB3D193025173CE4A842B1564E98519029DB64AFD54B195CF126D396A7BCB
       FCCCB3D193025173CE4A842B1564E98519029DB64AFD54B195CF126D396A7BCB
PCR 1: DE53615224E1C7C5BD2418705BB4A47D86308D2DFF021EFCEC86051BA1C6BDAF
       DE53615224E1C7C5BD2418705BB4A47D86308D2DFF021EFCEC86051BA1C6BDAF
PCR 2: 3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
       3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
PCR 3: 3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
       3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
PCR 4: 7400883517D37BF5391A15BB9CE2D7CBF18371202EF0F8ECE92E4085DFD30C4D
       7400883517D37BF5391A15BB9CE2D7CBF18371202EF0F8ECE92E4085DFD30C4D
PCR 5: 9F2211EF1E6B9B809B0463730A8707357CF13419D37ABD757B35AFE175159F10
       9F2211EF1E6B9B809B0463730A8707357CF13419D37ABD757B35AFE175159F10
PCR 6: 3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
       3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
PCR 7: E78DE8E6A07E945D97468E3C147D588134215CCD52A9A50F74CC6974831DF004
       E78DE8E6A07E945D97468E3C147D588134215CCD52A9A50F74CC6974831DF004
```