Trusted Execution Environment Provisioning (TEEP) Architecture
draft-ietf-teep-architecture-17

Abstract

A Trusted Execution Environment (TEE) is an environment that enforces that any code within that environment cannot be tampered with, and that any data used by such code cannot be read or tampered with by any code outside that environment. This architecture document motivates the design and standardization of a protocol for managing the lifecycle of trusted applications running inside such a TEE.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 21 October 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document.
Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction .................................. 3
2. Terminology .................................. 5
3. Use Cases .................................... 8
   3.1. Payment ................................ 8
   3.2. Authentication .......................... 8
   3.3. Internet of Things ..................... 8
   3.4. Confidential Cloud Computing .......... 8
4. Architecture .................................. 9
   4.1. System Components ....................... 9
   4.2. Multiple TEEs in a Device ............ 11
   4.3. Multiple TAMs and Relationship to TAs ... 13
   4.4. Untrusted Apps, Trusted Apps, and Personalization Data . 15
      4.4.1. Example: Application Delivery Mechanisms in Intel
      TrustZone .................................. 17
   4.4.2. Example: Application Delivery Mechanisms in Arm
5. Keys and Certificate Types ................. 19
   5.1. Trust Anchors in a TEEP Agent .......... 21
   5.2. Trust Anchors in a TEE .................. 21
   5.3. Trust Anchors in a TAM .................. 21
   5.4. Scalability ............................. 22
   5.5. Message Security ......................... 22
6. TEEP Broker ................................. 22
   6.1. Role of the TEEP Broker ............... 23
   6.2. TEEP Broker Implementation Consideration ... 23
      6.2.1. TEEP Broker APIs ................ 24
      6.2.2. TEEP Broker Distribution .......... 25
7. Attestation .................................. 25
8. Algorithm and Attestation Agility .......... 28
9. Security Considerations ..................... 28
   9.1. Broker Trust Model ..................... 28
   9.2. Data Protection ........................ 29
   9.3. Compromised REE ......................... 30
   9.4. CA Compromise or Expiry of CA Certificate .... 31
   9.5. Compromised TAM ........................ 31
   9.6. Malicious TA Removal ................... 31
   9.7. TEE Certificate Expiry and Renewal .... 32
   9.8. Keeping Secrets from the TAM .......... 33
   9.9. REE Privacy ............................. 33
1. Introduction

Applications executing in a device are exposed to many different attacks intended to compromise the execution of the application or reveal the data upon which those applications are operating. These attacks increase with the number of other applications on the device, with such other applications coming from potentially untrustworthy sources. The potential for attacks further increases with the complexity of features and applications on devices, and the unintended interactions among those features and applications. The danger of attacks on a system increases as the sensitivity of the applications or data on the device increases. As an example, exposure of emails from a mail client is likely to be of concern to its owner, but a compromise of a banking application raises even greater concerns.

The Trusted Execution Environment (TEE) concept is designed to let applications execute in a protected environment that enforces that any code within that environment cannot be tampered with, and that any data used by such code cannot be read or tampered with by any code outside that environment, including by a commodity operating system (if present). In a system with multiple TEEs, this also means that code in one TEE cannot be read or tampered with by code in another TEE.

This separation reduces the possibility of a successful attack on application components and the data contained inside the TEE. Typically, application components are chosen to execute inside a TEE because those application components perform security sensitive operations or operate on sensitive data. An application component running inside a TEE is referred to as a Trusted Application (TA), while an application running outside any TEE, i.e., in the Rich Execution Environment (REE), is referred to as an Untrusted Application. In the example of a banking application, code that relates to the authentication protocol could reside in a TA while the application logic including HTTP protocol parsing could be contained in the Untrusted Application. In addition, processing of credit card numbers or account balances could be done in a TA as it is sensitive data. The precise code split is ultimately a decision of the developer based on the assets he or she wants to protect according to the threat model.
TEEs are typically used in cases where a software or data asset needs to be protected from unauthorized entities that may include the owner (or owner) or possessor of a device. This situation arises for example in gaming consoles where anti-cheat protection is a concern, devices such as ATMs or IoT devices placed in locations where attackers might have physical access, cell phones or other devices used for mobile payments, and hosted cloud environments. Such environments can be thought of as hybrid devices where one user or administrator controls the REE and a different (remote) user or administrator controls a TEE in the same physical device. It may also be the case in some constrained devices that there is no REE (only a TEE) and there may be no local "user" per se, only a remote TEE administrator. For further discussion of such confidential computing use cases and threat model, see [CC-Overview] and [CC-Technical-Analysis].

TEEs use hardware enforcement combined with software protection to secure TAs and its data. TEEs typically offer a more limited set of services to TAs than is normally available to Untrusted Applications.

Not all TEEs are the same, however, and different vendors may have different implementations of TEEs with different security properties, different features, and different control mechanisms to operate on TAs. Some vendors may themselves market multiple different TEEs with different properties attuned to different markets. A device vendor may integrate one or more TEEs into their devices depending on market needs.

To simplify the life of TA developers interacting with TAs in a TEE, an interoperable protocol for managing TAs running in different TEEs of various devices is needed. This software update protocol needs to make sure that compatible trusted and untrusted components (if any) of an application are installed on the correct device. In this TEE ecosystem, there often arises a need for an external trusted party to verify the identity, claims, and rights of TA developers, devices, and their TEEs. This external trusted party is the Trusted Application Manager (TAM).

The Trusted Execution Environment Provisioning (TEEP) protocol addresses the following problems:

* An installer of an Untrusted Application that depends on a given TA wants to request installation of that TA in the device’s TEE so that the Untrusted Application can complete, but the TEE needs to verify whether such a TA is actually authorized to run in the TEE and consume potentially scarce TEE resources.
* A TA developer providing a TA whose code itself is considered confidential wants to determine security-relevant information of a device before allowing their TA to be provisioned to the TEE within the device. An example is the verification of the type of TEE included in a device and that it is capable of providing the security protections required.

* A TEE in a device wants to determine whether an entity that wants to manage a TA in the device is authorized to manage TAs in the TEE, and what TAs the entity is permitted to manage.

* A Device Administrator wants to determine if a TA exists (is installed) on a device (in the TEE), and if not, install the TA in the TEE.

* A Device Administrator wants to check whether a TA in a device’s TEE is the most up-to-date version, and if not, update the TA in the TEE.

* A Device Administrator wants to remove a TA from a device’s TEE if the TA developer is no longer maintaining that TA, when the TA has been revoked, or is not used for other reasons anymore (e.g., due to an expired subscription).

For TEEs that simply verify and load signed TA’s from an untrusted filesystem, classic application distribution protocols can be used without modification. The problems in the bullets above, on the other hand, require a new protocol, i.e., the TEEP protocol. The TEEP protocol is a solution for TEEs that can install and enumerate TAs in a TEE-secured location where another domain-specific protocol standard (e.g., [GSMA], [OTRP]) that meets the needs is not already in use.

2. Terminology

The following terms are used:

* App Store: An online location from which Untrusted Applications can be downloaded.

* Device: A physical piece of hardware that hosts one or more TEEs, often along with an REE.

* Device Administrator: An entity that is responsible for administration of a device, which could be the Device Owner. A Device Administrator has privileges on the device to install and remove Untrusted Applications and TAs, approve or reject Trust Anchors, and approve or reject TA developers, among possibly other
privileges on the device. A Device Administrator can manage the list of allowed TAMs by modifying the list of Trust Anchors on the device. Although a Device Administrator may have privileges and device-specific controls to locally administer a device, the Device Administrator may choose to remotely administer a device through a TAM.

* Device Owner: A device is always owned by someone. In some cases, it is common for the (primary) device user to also own the device, making the device user/owner also the Device Administrator. In enterprise environments it is more common for the enterprise to own the device, and any device user has no or limited administration rights. In this case, the enterprise appoints a Device Administrator that is not the device owner.

* Device User: A human being that uses a device. Many devices have a single device user. Some devices have a primary device user with other human beings as secondary device users (e.g., a parent allowing children to use their tablet or laptop). Other devices are not used by a human being and hence have no device user. Relates to Device Owner and Device Administrator.

* Personalization Data: A set of configuration data that is specific to the device or user. The Personalization Data may depend on the type of TEE, a particular TEE instance, the TA, and even the user of the device; an example of Personalization Data might be a secret symmetric key used by a TA to communicate with some service.

* Raw Public Key: A raw public key consists of only the algorithm identifier (type) of the key and the cryptographic public key material, such as the SubjectPublicKeyInfo structure of a PKIX certificate [RFC5280]. Other serialization formats that do not rely on ASN.1 may also be used.

* Rich Execution Environment (REE): An environment that is provided and governed by a typical OS (e.g., Linux, Windows, Android, iOS), potentially in conjunction with other supporting operating systems and hypervisors; it is outside of the TEE(s) managed by the TEEP protocol. This environment and applications running on it are considered untrusted (or more precisely, less trusted than a TEE).

* Trust Anchor: As defined in [RFC6024] and [I-D.ietf-suit-architecture], "A trust anchor represents an authoritative entity via a public key and associated data. The public key is used to verify digital signatures, and the associated data is used to constrain the types of information for which the trust anchor is authoritative." The Trust Anchor may be
a certificate, a raw public key or other structure, as appropriate. It can be a non-root certificate when it is a certificate.

* Trust Anchor Store: As defined in [RFC6024], "A trust anchor store is a set of one or more trust anchors stored in a device... A device may have more than one trust anchor store, each of which may be used by one or more applications." As noted in [I-D.ietf-suit-architecture], a Trust Anchor Store must resist modification against unauthorized insertion, deletion, and modification.

* Trusted Application (TA): An application (or, in some implementations, an application component) that runs in a TEE.

* Trusted Application Manager (TAM): An entity that manages Trusted Applications and other Trusted Components running in TEEs of various devices.

* Trusted Component: A set of code and/or data in a TEE managed as a unit by a Trusted Application Manager. Trusted Applications and Personalization Data are thus managed by being included in Trusted Components. Trusted OS code or trusted firmware can also be expressed as Trusted Components that a Trusted Component depends on.

* Trusted Component Developer: An entity that develops one or more Trusted Components.

* Trusted Component Signer: An entity that signs a Trusted Component with a key that a TEE will trust. The signer might or might not be the same entity as the Trusted Component Developer. For example, a Trusted Component might be signed (or re-signed) by a Device Administrator if the TEE will only trust the Device Administrator. A Trusted Component might also be encrypted, if the code is considered confidential.

* Trusted Execution Environment (TEE): An execution environment that enforces that only authorized code can execute within the TEE, and data used by that code cannot be read or tampered with by code outside the TEE. A TEE also generally has a device unique credential that cannot be cloned. There are multiple technologies that can be used to implement a TEE, and the level of security achieved varies accordingly. In addition, TEEs typically use an isolation mechanism between Trusted Applications to ensure that one TA cannot read, modify or delete the data and code of another TA.
Untrusted Application: An application running in an REE.  An Untrusted Application might depend on one or more TAs.

3. Use Cases

3.1. Payment

A payment application in a mobile device requires high security and trust in the hosting device. Payments initiated from a mobile device can use a Trusted Application to provide strong identification and proof of transaction.

For a mobile payment application, some biometric identification information could also be stored in a TEE. The mobile payment application can use such information for unlocking the device and for local identification of the user.

A trusted user interface (UI) may be used in a mobile device to prevent malicious software from stealing sensitive user input data. Such an implementation often relies on a TEE for providing access to peripherals, such as PIN input or a trusted display, so that the REE cannot observe or tamper with the user input or output.

3.2. Authentication

For better security of authentication, a device may store its keys and cryptographic libraries inside a TEE limiting access to cryptographic functions via a well-defined interface and thereby reducing access to keying material.

3.3. Internet of Things

Weak security in Internet of Things (IoT) devices has been posing threats to critical infrastructure, i.e., assets that are essential for the functioning of a society and economy. It is desirable that IoT devices can prevent malware from manipulating actuators (e.g., unlocking a door), or stealing or modifying sensitive data, such as authentication credentials in the device. A TEE can be the best way to implement such IoT security functions.

3.4. Confidential Cloud Computing

A tenant can store sensitive data, such as customer details or credit card numbers, in a TEE in a cloud computing server such that only the tenant can access the data, preventing the cloud hosting provider from accessing the data. A tenant can run TAs inside a server TEE for secure operation and enhanced data security. This provides benefits not only to tenants with better data security but also to
cloud hosting providers for reduced liability and increased cloud adoption.

4. Architecture

4.1. System Components

Figure 1 shows the main components in a typical device with an REE and a TEE. Full descriptions of components not previously defined are provided below. Interactions of all components are further explained in the following paragraphs.

* Trusted Component Signers and Device Administrators utilize the services of a TAM to manage TAs on devices. Trusted Component Signers do not directly interact with devices. Device Administrators may elect to use a TAM for remote administration of TAs instead of managing each device directly.

* Trusted Application Manager (TAM): A TAM is responsible for performing lifecycle management activity on Trusted Components on behalf of Trusted Component Signers and Device Administrators. This includes installation and deletion of Trusted Components, and may include, for example, over-the-air updates to keep Trusted Components up-to-date and clean up when Trusted Components should be removed. TAMS may provide services that make it easier for
Trusted Component Signers or Device Administrators to use the TAM’s service to manage multiple devices, although that is not required of a TAM.

The TAM performs its management of Trusted Components on the device through interactions with a device’s TEEP Broker, which relays messages between a TAM and a TEEP Agent running inside the TEE. TEEP authentication is performed between a TAM and a TEEP Agent.

As shown in Figure 1, the TAM cannot directly contact a TEEP Agent, but must wait for the TEEP Broker to contact the TAM requesting a particular service. This architecture is intentional in order to accommodate network and application firewalls that normally protect user and enterprise devices from arbitrary connections from external network entities.

A TAM may be publicly available for use by many Trusted Component Signers, or a TAM may be private, and accessible by only one or a limited number of Trusted Component Signers. It is expected that many enterprises, manufacturers, and network carriers will run their own private TAM.

A Trusted Component Signer or Device Administrator chooses a particular TAM based on whether the TAM is trusted by a device or set of devices. The TAM is trusted by a device if the TAM’s public key is, or chains up to, an authorized Trust Anchor in the device. A Trusted Component Signer or Device Administrator may run their own TAM, but the devices they wish to manage must include this TAM’s public key or certificate, or a certificate it chains up to, in the Trust Anchor Store.

A Trusted Component Signer or Device Administrator is free to utilize multiple TAMs. This may be required for managing Trusted Components on multiple different types of devices from different manufacturers, or mobile devices on different network carriers, since the Trust Anchor Store on these different devices may contain keys for different TAMs. A Device Administrator may be able to add their own TAM’s public key or certificate, or a certificate it chains up to, to the Trust Anchor Store on all their devices, overcoming this limitation.
Any entity is free to operate a TAM. For a TAM to be successful, it must have its public key or certificate installed in a device’s Trust Anchor Store. A TAM may set up a relationship with device manufacturers or network carriers to have them install the TAM’s keys in their device’s Trust Anchor Store. Alternatively, a TAM may publish its certificate and allow Device Administrators to install the TAM’s certificate in their devices as an after-market action.

* TEEP Broker: A TEEP Broker is an application component running in a Rich Execution Environment (REE) that enables the message protocol exchange between a TAM and a TEE in a device. A TEEP Broker does not process messages on behalf of a TEE, but merely is responsible for relaying messages from the TAM to the TEE, and for returning the TEE’s responses to the TAM. In devices with no REE (e.g., a microcontroller where all code runs in an environment that meets the definition of a Trusted Execution Environment in Section 2), the TEEP Broker would be absent and instead the TEEP protocol transport would be implemented inside the TEE itself.

* TEEP Agent: The TEEP Agent is a processing module running inside a TEE that receives TAM requests (typically relayed via a TEEP Broker that runs in an REE). A TEEP Agent in the TEE may parse requests or forward requests to other processing modules in a TEE, which is up to a TEE provider’s implementation. A response message corresponding to a TAM request is sent back to the TAM, again typically relayed via a TEEP Broker.

* Certification Authority (CA): A CA is an entity that issues digital certificates (especially X.509 certificates) and vouches for the binding between the data items in a certificate [RFC4949]. Certificates are then used for authenticating a device, a TAM, or a Trusted Component Signer, as discussed in Section 5. The CAs do not need to be the same; different CAs can be chosen by each TAM, and different device CAs can be used by different device manufacturers.

4.2. Multiple TEEs in a Device

Some devices might implement multiple TEEs. In these cases, there might be one shared TEEP Broker that interacts with all the TEEs in the device. However, some TEEs (for example, SGX [SGX]) present themselves as separate containers within memory without a controlling manager within the TEE. As such, there might be multiple TEEP Brokers in the REE, where each TEEP Broker communicates with one or more TEEs associated with it.
It is up to the REE and the Untrusted Applications how they select the correct TEEP Broker. Verification that the correct TA has been reached then becomes a matter of properly verifying TA attestations, which are unforgeable.

The multiple TEEP Broker approach is shown in the diagram below. For brevity, TEEP Broker 2 is shown interacting with only one TAM and Untrusted Application and only one TEE, but no such limitations are intended to be implied in the architecture.

![Diagram of TEEP Architecture with multiple TEEs](image-url)

Figure 2: Notional Architecture of TEEP with multiple TEEs
In the diagram above, TEEP Broker 1 controls interactions with the TAs in TEE-1, and TEEP Broker 2 controls interactions with the TAs in TEE-2. This presents some challenges for a TAM in completely managing the device, since a TAM may not interact with all the TEEP Brokers on a particular platform. In addition, since TEEs may be physically separated, with wholly different resources, there may be no need for TEEP Brokers to share information on installed Trusted Components or resource usage.

4.3. Multiple TAMs and Relationship to TAs

As shown in Figure 2, a TEEP Broker provides communication between one or more TEEP Agents and one or more TAMs. The selection of which TAM to interact with might be made with or without input from an Untrusted Application, but is ultimately the decision of a TEEP Agent.

A TEEP Agent is assumed to be able to determine, for any given Trusted Component, whether that Trusted Component is installed (or minimally, is running) in a TEE with which the TEEP Agent is associated.

Each Trusted Component is digitally signed, protecting its integrity, and linking the Trusted Component back to the Trusted Component Signer. The Trusted Component Signer is often the Trusted Component Developer, but in some cases might be another party such as a Device Administrator or other party to whom the code has been licensed (in which case the same code might be signed by multiple licensees and distributed as if it were different TAs).

A Trusted Component Signer selects one or more TAMs and communicates the Trusted Component(s) to the TAM. For example, the Trusted Component Signer might choose TAMs based upon the markets into which the TAM can provide access. There may be TAMs that provide services to specific types of devices, or device operating systems, or specific geographical regions or network carriers. A Trusted Component Signer may be motivated to utilize multiple TAMs in order to maximize market penetration and availability on multiple types of devices. This means that the same Trusted Component will often be available through multiple TAMs.

When the developer of an Untrusted Application that depends on a Trusted Component publishes the Untrusted Application to an app store or other app repository, the developer optionally binds the Untrusted Application with a manifest that identifies what TAMs can be contacted for the Trusted Component. In some situations, a Trusted Component may only be available via a single TAM - this is likely the case for enterprise applications or Trusted Component Signers serving
a closed community. For broad public apps, there will likely be multiple TAMs in the Untrusted Application’s manifest – one servicing one brand of mobile device and another servicing a different manufacturer, etc. Because different devices and different manufacturers trust different TAMs, the manifest can include multiple TAMs that support the required Trusted Component.

When a TEEP Broker receives a request (see the RequestTA API in Section 6.2.1) from an Untrusted Application to install a Trusted Component, a list of TAM URIs may be provided for that Trusted Component, and the request is passed to the TEEP Agent. If the TEEP Agent decides that the Trusted Component needs to be installed, the TEEP Agent selects a single TAM URI that is consistent with the list of trusted TAMs provisioned in the TEEP Agent, invokes the HTTP transport for TEEP to connect to the TAM URI, and begins a TEEP protocol exchange. When the TEEP Agent subsequently receives the Trusted Component to install and the Trusted Component’s manifest indicates dependencies on any other trusted components, each dependency can include a list of TAM URIs for the relevant dependency. If such dependencies exist that are prerequisites to install the Trusted Component, then the TEEP Agent recursively follows the same procedure for each dependency that needs to be installed or updated, including selecting a TAM URI that is consistent with the list of trusted TAMs provisioned on the device, and beginning a TEEP exchange. If multiple TAM URIs are considered trusted, only one needs to be contacted and they can be attempted in some order until one responds.

Separate from the Untrusted Application’s manifest, this framework relies on the use of the manifest format in [I-D.ietf-suit-manifest] for expressing how to install a Trusted Component, as well as any dependencies on other TEE components and versions. That is, dependencies from Trusted Components on other Trusted Components can be expressed in a SUIT manifest, including dependencies on any other TAs, trusted OS code (if any), or trusted firmware. Installation steps can also be expressed in a SUIT manifest.

For example, TEEs compliant with GlobalPlatform [GPTEE] may have a notion of a "security domain" (which is a grouping of one or more TAs installed on a device, that can share information within such a group) that must be created and into which one or more TAs can then be installed. It is thus up to the SUIT manifest to express a dependency on having such a security domain existing or being created first, as appropriate.
Updating a Trusted Component may cause compatibility issues with any Untrusted Applications or other components that depend on the updated Trusted Component, just like updating the OS or a shared library could impact an Untrusted Application. Thus, an implementation needs to take into account such issues.

4.4. Untrusted Apps, Trusted Apps, and Personalization Data

In TEEP, there is an explicit relationship and dependence between an Untrusted Application in an REE and one or more TAs in a TEE, as shown in Figure 2. For most purposes, an Untrusted Application that uses one or more TAs in a TEE appears no different from any other Untrusted Application in the REE. However, the way the Untrusted Application and its corresponding TAs are packaged, delivered, and installed on the device can vary. The variations depend on whether the Untrusted Application and TA are bundled together or are provided separately, and this has implications to the management of the TAs in a TEE. In addition to the Untrusted Application and TA(s), the TA(s) and/or TEE may also require additional data to personalize the TA to the device or a user. Implementations must support encryption to preserve the confidentiality and integrity of such Personalized Data, which may potentially contain sensitive data. Other than the requirement to support confidentiality and integrity protection, the TEEP architecture places no limitations or requirements on the Personalization Data.

There are multiple possible cases for bundling of an Untrusted Application, TA(s), and Personalization Data. Such cases include (possibly among others):

1. The Untrusted Application, TA(s), and Personalization Data are all bundled together in a single package by a Trusted Component Signer and either provided to the TEEP Broker through the TAM, or provided separately (with encrypted Personalization Data), with key material needed to decrypt and install the Personalization Data and TA provided by a TAM.

2. The Untrusted Application and the TA(s) are bundled together in a single package, which a TAM or a publicly accessible app store maintains, and the Personalization Data is separately provided by the Personalization Data provider’s TAM.

3. All components are independent packages. The Untrusted Application is installed through some independent or device-specific mechanism, and one or more TAMs provide (directly or indirectly by reference) the TA(s) and Personalization Data.
4. The TA(s) and Personalization Data are bundled together into a package provided by a TAM, while the Untrusted Application is installed through some independent or device-specific mechanism such as an app store.

5. Encrypted Personalization Data is bundled into a package distributed with the Untrusted Application, while the TA(s) and key material needed to decrypt and install the Personalization Data are in a separate package provided by a TAM.

The TEEP protocol can treat each TA, any dependencies the TA has, and Personalization Data as separate Trusted Components with separate installation steps that are expressed in SUIT manifests, and a SUIT manifest might contain or reference multiple binaries (see [I-D.ietf-suit-manifest] for more details). The TEEP Agent is responsible for handling any installation steps that need to be performed inside the TEE, such as decryption of private TA binaries or Personalization Data.

In order to better understand these cases, it is helpful to review actual implementations of TEEs and their application delivery mechanisms.

4.4.1. Example: Application Delivery Mechanisms in Intel SGX

In Intel Software Guard Extensions (SGX), the Untrusted Application and TA are typically bundled into the same package (Case 2). The TA exists in the package as a shared library (.so or .dll). The Untrusted Application loads the TA into an SGX enclave when the Untrusted Application needs the TA. This organization makes it easy to maintain compatibility between the Untrusted Application and the TA, since they are updated together. It is entirely possible to create an Untrusted Application that loads an external TA into an SGX enclave, and use that TA (Cases 3-5). In this case, the Untrusted Application would require a reference to an external file or download such a file dynamically, place the contents of the file into memory, and load that as a TA. Obviously, such file or downloaded content must be properly formatted and signed for it to be accepted by the SGX TEE.

In SGX, any Personalization Data is normally loaded into the SGX enclave (the TA) after the TA has started. Although it is possible with SGX to include the Untrusted Application in an encrypted package along with Personalization Data (Cases 1 and 5), there are no instances of this known to be in use at this time, since such a construction would require a special installation program and SGX TA (which might or might not be the TEEP Agent itself based on the implementation) to receive the encrypted package, decrypt it,
separate it into the different elements, and then install each one. This installation is complex because the Untrusted Application decrypted inside the TEE must be passed out of the TEE to an installer in the REE which would install the Untrusted Application. Finally, the Personalization Data would need to be sent out of the TEE (encrypted in an SGX enclave-to-enclave manner) to the REE’s installation app, which would pass this data to the installed Untrusted Application, which would in turn send this data to the SGX enclave (TA). This complexity is due to the fact that each SGX enclave is separate and does not have direct communication to other SGX enclaves.

As long as signed files (TAs and/or Personalization Data) are installed into an untrusted filesystem and trust is verified by the TEE at load time, classic distribution mechanisms can be used. Some uses of SGX, however, allow a model where a TA can be dynamically installed into an SGX enclave that provides a runtime platform. The TEEP protocol can be used in such cases, where the runtime platform could include a TEEP Agent.

4.4.2. Example: Application Delivery Mechanisms in Arm TrustZone

In Arm TrustZone [TrustZone] for A-class devices, the Untrusted Application and TA may or may not be bundled together. This differs from SGX since in TrustZone the TA lifetime is not inherently tied to a specific Untrusted Application process lifetime as occurs in SGX. A TA is loaded by a trusted OS running in the TEE such as a GlobalPlatform [GPTEE] compliant TEE, where the trusted OS is separate from the OS in the REE. Thus Cases 2-4 are equally applicable. In addition, it is possible for TAs to communicate with each other without involving any Untrusted Application, and so the complexity of Cases 1 and 5 are lower than in the SGX example, though still more complex than Cases 2-4.

A trusted OS running in the TEE (e.g., OP-TEE) that supports loading and verifying signed TAs from an untrusted filesystem can, like SGX, use classic file distribution mechanisms. If secure TA storage is used (e.g., a Replay-Protected Memory Block device) on the other hand, the TEEP protocol can be used to manage such storage.

4.5. Entity Relations

This architecture leverages asymmetric cryptography to authenticate a device to a TAM. Additionally, a TEEP Agent in a device authenticates a TAM. The provisioning of Trust Anchors to a device may be different from one use case to the other. A Device Administrator may want to have the capability to control what TAs are allowed. A device manufacturer enables verification by one or more
TAMs and by Trusted Component Signers; it may embed a list of default Trust Anchors into the TEEP Agent and TEE for TAM trust verification and TA signature verification.

(App Developers) (App Store) (TAM) (Device with TEE) (CAs)

| (Embedded TEE cert) <-- | Get an app cert --------------- | Get a TAM cert -----

1. Build two apps:

(a) Untrusted App - 2a. Supply --> 3. Install ---->

(b) TA -- 2b. Supply --------> 4. Messaging-->

Figure 3: Example Developer Experience

Figure 3 shows an example where the same developer builds and signs two applications: (a) an Untrusted Application; (b) a TA that provides some security functions to be run inside a TEE. This example assumes that the developer, the TEE, and the TAM have previously been provisioned with certificates.

At step 1, the developer authors the two applications.

At step 2, the developer uploads the Untrusted Application (2a) to an Application Store. In this example, the developer is also the Trusted Component Signer, and so generates a signed TA. The developer can then either bundle the signed TA with the Untrusted Application, or the developer can provide a signed Trusted Component containing the TA to a TAM that will be managing the TA in various devices.

At step 3, a user will go to an Application Store to download the Untrusted Application (where the arrow indicates the direction of data transfer).

At step 4, since the Untrusted Application depends on the TA, installing the Untrusted Application will trigger TA installation via communication with a TAM. The TEEP Agent will interact with the TAM via a TEEP Broker that facilitates communications between the TAM and the TEEP Agent.
Some Trusted Component installation implementations might ask for a user’s consent. In other implementations, a Device Administrator might choose what Untrusted Applications and related Trusted Components to be installed. A user consent flow is out of scope of the TEEP architecture.

The main components of the TEEP protocol consist of a set of standard messages created by a TAM to deliver Trusted Component management commands to a device, and device attestation and response messages created by a TEE that responds to a TAM’s message.

It should be noted that network communication capability is generally not available in TAs in today’s TEE-powered devices. Consequently, Trusted Applications generally rely on a broker in the REE to provide access to network functionality in the REE. A broker does not need to know the actual content of messages to facilitate such access.

Similarly, since the TEEP Agent runs inside a TEE, the TEEP Agent generally relies on a TEEP Broker in the REE to provide network access, and relay TAM requests to the TEEP Agent and relay the responses back to the TAM.

5. Keys and Certificate Types

This architecture leverages the following credentials, which allow achieving end-to-end security between a TAM and a TEEP Agent.

Figure 4 summarizes the relationships between various keys and where they are stored. Each public/private key identifies a Trusted Component Signer, TAM, or TEE, and gets a certificate that chains up to some trust anchor. A list of trusted certificates is used to check a presented certificate against.

Different CAs can be used for different types of certificates. TEEP messages are always signed, where the signer key is the message originator’s private key, such as that of a TAM or a TEE. In addition to the keys shown in Figure 4, there may be additional keys used for attestation or encryption. Refer to the RATS Architecture [I-D.ietf-rats-architecture] for more discussion.
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Cardinality &amp; Location of Private Key</th>
<th>Private Key Signs</th>
<th>Location of Trust Anchor Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticating TAM</td>
<td>1 per TAM</td>
<td>TEEP requests</td>
<td>TEEP Agent</td>
</tr>
<tr>
<td>TEEP Agent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code Signing</td>
<td>1 per Trusted Component Signer</td>
<td>TA binary</td>
<td>TEE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Signature Keys

Note that Personalization Data is not included in the table above. The use of Personalization Data is dependent on how TAs are used and what their security requirements are.

TEEP requests from a TAM to a TEEP Agent are signed with the TAM private key (for authentication and integrity protection). Personalization Data and TA binaries can be encrypted with a key that is established with a content-encryption key established with the TEE public key (to provide confidentiality). Conversely, TEEP responses from a TEEP Agent to a TAM can be signed with the TEE private key.

The TEE key pair and certificate are thus used for authenticating the TEE to a remote TAM, and for sending private data to the TEE. Often, the key pair is burned into the TEE by the TEE manufacturer and the key pair and its certificate are valid for the expected lifetime of the TEE. A TAM provider is responsible for configuring the TAM’s Trust Anchor Store with the manufacturer certificates or CAs that are used to sign TEE keys. This is discussed further in Section 5.3 below. Typically the same key TEE pair is used for both signing and encryption, though separate key pairs might also be used in the future, as the joint security of encryption and signature with a single key remains to some extent an open question in academic cryptography.

The TAM key pair and certificate are used for authenticating a TAM to a remote TEE, and for sending private data to the TAM (separate key pairs for authentication vs. encryption could also be used in the future). A TAM provider is responsible for acquiring a certificate from a CA that is trusted by the TEEs it manages. This is discussed further in Section 5.1 below.
The Trusted Component Signer key pair and certificate are used to sign Trusted Components that the TEE will consider authorized to execute. TEEs must be configured with the certificates or keys that it considers authorized to sign TAs that it will execute. This is discussed further in Section 5.2 below.

5.1. Trust Anchors in a TEEP Agent

A TEEP Agent’s Trust Anchor Store contains a list of Trust Anchors, which are typically CA certificates that sign various TAM certificates. The list is typically preloaded at manufacturing time, and can be updated using the TEEP protocol if the TEE has some form of "Trust Anchor Manager TA" that has Trust Anchors in its configuration data. Thus, Trust Anchors can be updated similarly to the Personalization Data for any other TA.

When Trust Anchor update is carried out, it is imperative that any update must maintain integrity where only an authentic Trust Anchor list from a device manufacturer or a Device Administrator is accepted. Details are out of scope of the architecture and can be addressed in a protocol document.

Before a TAM can begin operation in the marketplace to support a device with a particular TEE, it must be able to get its raw public key, or its certificate, or a certificate it chains up to, listed in the Trust Anchor Store of the TEEP Agent.

5.2. Trust Anchors in a TEE

The Trust Anchor Store in a TEE contains a list of Trust Anchors (raw public keys or certificates) that are used to determine whether TA binaries are allowed to execute by checking if their signatures can be verified. The list is typically preloaded at manufacturing time, and can be updated using the TEEP protocol if the TEE has some form of "Trust Anchor Manager TA" that has Trust Anchors in its configuration data. Thus, Trust Anchors can be updated similarly to the Personalization Data for any other TA, as discussed in Section 5.1.

5.3. Trust Anchors in a TAM

The Trust Anchor Store in a TAM consists of a list of Trust Anchors, which are certificates that sign various device TEE certificates. A TAM will accept a device for Trusted Component management if the TEE in the device uses a TEE certificate that is chained to a certificate or raw public key that the TAM trusts, is contained in an allow list, is not found on a block list, and/or fulfills any other policy criteria.
5.4. Scalability

This architecture uses a PKI (including self-signed certificates). Trust Anchors exist on the devices to enable the TEEP Agent to authenticate TAMs and the TEE to authenticate Trusted Component Signers, and TAMs use Trust Anchors to authenticate TEEP Agents. When a PKI is used, many intermediate CA certificates can chain to a root certificate, each of which can issue many certificates. This makes the protocol highly scalable. New factories that produce TEEs can join the ecosystem. In this case, such a factory can get an intermediate CA certificate from one of the existing roots without requiring that TAMs are updated with information about the new device factory. Likewise, new TAMs can join the ecosystem, providing they are issued a TAM certificate that chains to an existing root whereby existing TEEs will be allowed to be personalized by the TAM without requiring changes to the TEE itself. This enables the ecosystem to scale, and avoids the need for centralized databases of all TEEs produced or all TAMs that exist or all Trusted Component Signers that exist.

5.5. Message Security

Messages created by a TAM are used to deliver Trusted Component management commands to a device, and device attestation and messages are created by the device TEE to respond to TAM messages. These messages are signed end-to-end between a TEEP Agent and a TAM. Confidentiality is provided by encrypting sensitive payloads (such as Personalization Data and attestation evidence), rather than encrypting the messages themselves. Using encrypted payloads is important to ensure that only the targeted device TEE or TAM is able to decrypt and view the actual content.

6. TEEP Broker

A TEE and TAs often do not have the capability to directly communicate outside of the hosting device. For example, GlobalPlatform [GPTEE] specifies one such architecture. This calls for a software module in the REE world to handle network communication with a TAM.

A TEEP Broker is an application component running in the REE of the device or an SDK that facilitates communication between a TAM and a TEE. It also provides interfaces for Untrusted Applications to query and trigger installation of Trusted Components that the application needs to use.
An Untrusted Application might communicate with a TEEP Broker at runtime to trigger Trusted Component installation itself, or an Untrusted Application might simply have a metadata file that describes the Trusted Components it depends on and the associated TAM(s) for each Trusted Component, and an REE Application Installer can inspect this application metadata file and invoke the TEEP Broker to trigger Trusted Component installation on behalf of the Untrusted Application without requiring the Untrusted Application to run first.

6.1. Role of the TEEP Broker

A TEEP Broker abstracts the message exchanges with a TEE in a device. The input data is originated from a TAM or the first initialization call to trigger a Trusted Component installation.

The Broker doesn’t need to parse TEEP message content received from a TAM that should be processed by a TEE (see the ProcessTeepMessage API in Section 6.2.1). When a device has more than one TEE, one TEEP Broker per TEE could be present in the REE or a common TEEP Broker could be used by multiple TEEs where the transport protocol (e.g., [I-D.ietf-teep-otrp-over-http]) allows the TEEP Broker to distinguish which TEE is relevant for each message from a TAM.

The TEEP Broker interacts with a TEEP Agent inside a TEE, and relays the response messages generated from the TEEP Agent back to the TAM.

The Broker only needs to return a (transport) error message to the TAM if the TEE is not reachable for some reason. Other errors are represented as TEEP response messages returned from the TEE which will then be passed to the TAM.

6.2. TEEP Broker Implementation Consideration

As depicted in Figure 5, there are multiple ways in which a TEEP Broker can be implemented, with more or fewer layers being inside the TEE. For example, in model A, the model with the smallest TEE footprint, only the TEEP implementation is inside the TEE, whereas the TEEP/HTTP implementation is in the TEEP Broker outside the TEE.
Figure 5: TEEP Broker Models

In other models, additional layers are moved into the TEE, increasing the TEE footprint, with the Broker either containing or calling the topmost protocol layer outside of the TEE. An implementation is free to choose any of these models.

TEEP Broker implementers should consider methods of distribution, scope and concurrency on devices and runtime options.

6.2.1. TEEP Broker APIs

The following conceptual APIs exist from a TEEP Broker to a TEEP Agent:

1. RequestTA: A notification from an REE application (e.g., an installer, or an Untrusted Application) that it depends on a given Trusted Component, which may or may not already be installed in the TEE.
2. **UnrequestTA**: A notification from an REE application (e.g., an installer, or an Untrusted Application) that it no longer depends on a given Trusted Component, which may or may not already be installed in the TEE. For example, if the Untrusted Application is uninstalled, the uninstaller might invoke this conceptual API.

3. **ProcessTeepMessage**: A message arriving from the network, to be delivered to the TEEP Agent for processing.

4. **RequestPolicyCheck**: A hint (e.g., based on a timer) that the TEEP Agent may wish to contact the TAM for any changes, without the device itself needing any particular change.

5. **ProcessError**: A notification that the TEEP Broker could not deliver an outbound TEEP message to a TAM.

For comparison, similar APIs may exist on the TAM side, where a Broker may or may not exist, depending on whether the TAM uses a TEE or not:

1. **ProcessConnect**: A notification that a new TEEP session is being requested by a TEEP Agent.

2. **ProcessTeepMessage**: A message arriving on an existing TEEP session, to be delivered to the TAM for processing.

For further discussion on these APIs, see [I-D.ietf-teep-otrp-over-http].

6.2.2. **TEEP Broker Distribution**

The Broker installation is commonly carried out at OEM time. A user can dynamically download and install a Broker on-demand.

7. **Attestation**

Attestation is the process through which one entity (an Attester) presents "evidence", in the form of a series of claims, to another entity (a Verifier), and provides sufficient proof that the claims are true. Different Verifiers may require different degrees of confidence in attestation proofs and not all attestations are acceptable to every verifier. A third entity (a Relying Party) can then use "attestation results", in the form of another series of claims, from a Verifier to make authorization decisions. (See [I-D.ietf-rats-architecture] for more discussion.)
In TEEP, as depicted in Figure 6, the primary purpose of an attestation is to allow a device (the Attester) to prove to a TAM (the Relying Party) that a TEE in the device has particular properties, was built by a particular manufacturer, and/or is executing a particular TA. Other claims are possible; TEEP does not limit the claims that may appear in evidence or attestation results, but defines a minimal set of attestation result claims required for TEEP to operate properly. Extensions to these claims are possible. Other standards or groups may define the format and semantics of extended claims.

As of the writing of this specification, device and TEE attestations have not been standardized across the market. Different devices, manufacturers, and TEEs support different attestation protocols. In order for TEEP to be inclusive, it is agnostic to the format of evidence, allowing proprietary or standardized formats to be used between a TEE and a verifier (which may or may not be colocated in the TAM), as long as the format supports encryption of any information that is considered sensitive.

However, it should be recognized that not all Verifiers may be able to process all proprietary forms of attestation evidence. Similarly, the TEEP protocol is agnostic as to the format of attestation results, and the protocol (if any) used between the TAM and a verifier, as long as they convey at least the required set of claims in some format. Note that the respective attestation algorithms are not defined in the TEEP protocol itself; see [I-D.ietf-rats-architecture] and [I-D.ietf-teep-protocol] for more discussion.

There are a number of considerations that need to be considered when appraising evidence provided by a TEE, including:

* What security measures a manufacturer takes when provisioning keys into devices/TEEs;

* What hardware and software components have access to the attestation keys of the TEE;
* The source or local verification of claims within an attestation
  prior to a TEE signing a set of claims;

* The level of protection afforded to attestation keys against
  exfiltration, modification, and side channel attacks;

* The limitations of use applied to TEE attestation keys;

* The processes in place to discover or detect TEE breaches; and

* The revocation and recovery process of TEE attestation keys.

Some TAMs may require additional claims in order to properly
authorize a device or TEE. The specific format for these additional
claims are outside the scope of this specification, but the TEEP
protocol allows these additional claims to be included in the
attestation messages.

For more discussion of the attestation and appraisal process, see the
RATS Architecture [I-D.ietf-rats-architecture].

The following information is required for TEEP attestation:

* Device Identifying Information: Attestation information may need
to uniquely identify a device to the TAM. Unique device
identification allows the TAM to provide services to the device,
such as managing installed TAs, and providing subscriptions to
services, and locating device-specific keying material to
communicate with or authenticate the device. In some use cases it
may be sufficient to identify only the class of the device. The
security and privacy requirements regarding device identification
will vary with the type of TA provisioned to the TEE.

* TEE Identifying Information: The type of TEE that generated this
attestation must be identified. This includes version
identification information for hardware, firmware, and software
version of the TEE, as applicable by the TEE type. TEE
manufacturer information for the TEE is required in order to
disambiguate the same TEE type created by different manufacturers
and address considerations around manufacturer provisioning,
keying and support for the TEE.

* Freshness Proof: A claim that includes freshness information must
be included, such as a nonce or timestamp.
8. Algorithm and Attestation Agility

RFC 7696 [RFC7696] outlines the requirements to migrate from one mandatory-to-implement cryptographic algorithm suite to another over time. This feature is also known as crypto agility. Protocol evolution is greatly simplified when crypto agility is considered during the design of the protocol. In the case of the TEEP protocol the diverse range of use cases, from trusted app updates for smartphones and tablets to updates of code on higher-end IoT devices, creates the need for different mandatory-to-implement algorithms already from the start.

Crypto agility in TEEP concerns the use of symmetric as well as asymmetric algorithms. In the context of TEEP, symmetric algorithms are used for encryption and integrity protection of TA binaries and Personalization Data whereas the asymmetric algorithms are used for signing messages and managing symmetric keys.

In addition to the use of cryptographic algorithms in TEEP, there is also the need to make use of different attestation technologies. A device must provide techniques to inform a TAM about the attestation technology it supports. For many deployment cases it is more likely for the TAM to support one or more attestation techniques whereas the device may only support one.

9. Security Considerations

9.1. Broker Trust Model

The architecture enables the TAM to communicate, via a TEEP Broker, with the device’s TEE to manage Trusted Components. Since the TEEP Broker runs in a potentially vulnerable REE, the TEEP Broker could, however, be (or be infected by) malware. As such, all TAM messages are signed and sensitive data is encrypted such that the TEEP Broker cannot modify or capture sensitive data, but the TEEP Broker can still conduct DoS attacks as discussed in Section 9.3.

A TEEP Agent in a TEE is responsible for protecting against potential attacks from a compromised TEEP Broker or rogue malware in the REE. A rogue TEEP Broker might send corrupted data to the TEEP Agent, or launch a DoS attack by sending a flood of TEEP protocol requests, or simply drop or delay notifications to a TEE. The TEEP Agent validates the signature of each TEEP protocol request and checks the signing certificate against its Trust Anchors. To mitigate DoS attacks, it might also add some protection scheme such as a threshold on repeated requests or number of TAs that can be installed.
Some implementations might rely on (due to lack of any available alternative) the use of an untrusted timer or other event to call the RequestPolicyCheck API (Section 6.2.1), which means that a compromised REE can cause a TEE to not receive policy changes and thus be out of date with respect to policy. The same can potentially be done by any other man-in-the-middle simply by blocking communication with a TAM. Ultimately such outdated compliance could be addressed by using attestation in secure communication, where the attestation evidence reveals what state the TEE is in, so that communication (other than remediation such as via TEEP) from an out-of-compliance TEE can be rejected.

Similarly, in most implementations the REE is involved in the mechanics of installing new TAs. However, the authority for what TAs are running in a given TEE is between the TEEP Agent and the TAM. While a TEEP Broker broker can in effect make suggestions, it cannot decide or enforce what runs where. The TEEP Broker can also control which TEE a given installation request is directed at, but a TEEP Agent will only accept TAs that are actually applicable to it and where installation instructions are received by a TAM that it trusts.

The authorization model for the UnrequestTA operation is, however, weaker in that it expresses the removal of a dependency from an application that was untrusted to begin with. This means that a compromised REE could remove a valid dependency from an Untrusted Application on a TA. Normal REE security mechanisms should be used to protect the REE and Untrusted Applications.

9.2. Data Protection

It is the responsibility of the TAM to protect data on its servers. Similarly, it is the responsibility of the TEE implementation to provide protection of data against integrity and confidentiality attacks from outside the TEE. TEEs that provide isolation among TAs within the TEE are likewise responsible for protecting TA data against the REE and other TAs. For example, this can be used to protect one user’s or tenant’s data from compromise by another user or tenant, even if the attacker has TAs.

The protocol between TEEP Agents and TAMs similarly is responsible for securely providing integrity and confidentiality protection against adversaries between them. It is a design choice at what layers to best provide protection against network adversaries. As discussed in Section 6, the transport protocol and any security mechanism associated with it (e.g., the Transport Layer Security protocol) under the TEEP protocol may terminate outside a TEE. If it does, the TEEP protocol itself must provide integrity protection and confidentiality protection to secure data end-to-end. For example,
confidentiality protection for payloads may be provided by utilizing encrypted TA binaries and encrypted attestation information. See [I-D.ietf-teep-protocol] for how a specific solution addresses the design question of how to provide integrity and confidentiality protection.

9.3. Compromised REE

It is possible that the REE of a device is compromised. We have already seen examples of attacks on the public Internet with billions of compromised devices being used to mount DDoS attacks. A compromised REE can be used for such an attack but it cannot tamper with the TEE’s code or data in doing so. A compromised REE can, however, launch DoS attacks against the TEE.

The compromised REE may terminate the TEEP Broker such that TEEP transactions cannot reach the TEE, or might drop, replay, or delay messages between a TAM and a TEEP Agent. However, while a DoS attack cannot be prevented, the REE cannot access anything in the TEE if the TEE is implemented correctly. Some TEEs may have some watchdog scheme to observe REE state and mitigate DoS attacks against it but most TEEs don’t have such a capability.

In some other scenarios, the compromised REE may ask a TEEP Broker to make repeated requests to a TEEP Agent in a TEE to install or uninstall a Trusted Component. An installation or uninstallation request constructed by the TEEP Broker or REE will be rejected by the TEEP Agent because the request won’t have the correct signature from a TAM to pass the request signature validation.

This can become a DoS attack by exhausting resources in a TEE with repeated requests. In general, a DoS attack threat exists when the REE is compromised, and a DoS attack can happen to other resources. The TEEP architecture doesn’t change this.

A compromised REE might also request initiating the full flow of installation of Trusted Components that are not necessary. It may also repeat a prior legitimate Trusted Component installation request. A TEEP Agent implementation is responsible for ensuring that it can recognize and decline such repeated requests. It is also responsible for protecting the resource usage allocated for Trusted Component management.
9.4. CA Compromise or Expiry of CA Certificate

A root CA for TAM certificates might get compromised or its certificate might expire, or a Trust Anchor other than a root CA certificate may also expire or be compromised. TEEs are responsible for validating the entire TAM certificate path, including the TAM certificate and any intermediate certificates up to the root certificate. See Section 6 of [RFC5280] for details. Such validation generally includes checking for certificate revocation, but certificate status check protocols may not scale down to constrained devices that use TEEP.

To address the above issues, a certificate path update mechanism is expected from TAM operators, so that the TAM can get a new certificate path that can be validated by a TEEP Agent. In addition, the Trust Anchor in the TEEP Agent’s Trust Anchor Store may need to be updated. To address this, some TEE Trust Anchor update mechanism is expected from device OEMs, such as using the TEEP protocol to distribute new Trust Anchors.

Similarly, a root CA for TEE certificates might get compromised or its certificate might expire, or a Trust Anchor other than a root CA certificate may also expire or be compromised. TAMs are responsible for validating the entire TEE certificate path, including the TEE certificate and any intermediate certificates up to the root certificate. Such validation includes checking for certificate revocation.

If a TEE certificate path validation fails, the TEE might be rejected by a TAM, subject to the TAM’s policy. To address this, some certificate path update mechanism is expected from device OEMs, so that the TEE can get a new certificate path that can be validated by a TAM. In addition, the Trust Anchor in the TAM’s Trust Anchor Store may need to be updated.

9.5. Compromised TAM

Device TEEs are responsible for validating the supplied TAM certificates to determine that the TAM is trustworthy.

9.6. Malicious TA Removal

It is possible that a rogue developer distributes a malicious Untrusted Application and intends to get a malicious TA installed. Such a TA might be able to escape from malware detection by the REE, or access trusted resources within the TEE (but could not access other TEEs, or access other TA’s if the TEE provides isolation between TAs).
It is the responsibility of the TAM to not install malicious TAs in the first place. The TEEP architecture allows a TEEP Agent to decide which TAMS it trusts via Trust Anchors, and delegates the TA authenticity check to the TAMS it trusts.

It may happen that a TA was previously considered trustworthy but is later found to be buggy or compromised. In this case, the TAM can initiate the removal of the TA by notifying devices to remove the TA (and potentially the REE or Device Owner to remove any Untrusted Application that depend on the TA). If the TAM does not currently have a connection to the TEEP Agent on a device, such a notification would occur the next time connectivity does exist. That is, to recover, the TEEP Agent must be able to reach out to the TAM, for example whenever the RequestPolicyCheck API (Section 6.2.1) is invoked by a timer or other event.

Furthermore the policy in the Verifier in an attestation process can be updated so that any evidence that includes the malicious TA would result in an attestation failure. There is, however, a time window during which a malicious TA might be able to operate successfully, which is the validity time of the previous attestation result. For example, if the Verifier in Figure 6 is updated to treat a previously valid TA as no longer trustworthy, any attestation result it previously generated saying that the TA is valid will continue to be used until the attestation result expires. As such, the TAM’s Verifier should take into account the acceptable time window when generating attestation results. See [I-D.ietf-rats-architecture] for further discussion.

9.7. TEE Certificate Expiry and Renewal

TEE device certificates are expected to be long lived, longer than the lifetime of a device. A TAM certificate usually has a moderate lifetime of 2 to 5 years. A TAM should get renewed or rekeyed certificates. The root CA certificates for a TAM, which are embedded into the Trust Anchor Store in a device, should have long lifetimes that don’t require device Trust Anchor updates. On the other hand, it is imperative that OEMs or device providers plan for support of Trust Anchor update in their shipped devices.

For those cases where TEE devices are given certificates for which no good expiration date can be assigned the recommendations in Section 4.1.2.5 of [RFC5280] are applicable.
9.8. Keeping Secrets from the TAM

In some scenarios, it is desirable to protect the TA binary or Personalization Data from being disclosed to the TAM that distributes them. In such a scenario, the files can be encrypted end-to-end between a Trusted Component Signer and a TEE. However, there must be some means of provisioning the decryption key into the TEE and/or some means of the Trusted Component Signer securely learning a public key of the TEE that it can use to encrypt. The Trusted Component Signer cannot necessarily even trust the TAM to report the correct public key of a TEE for use with encryption, since the TAM might instead provide the public key of a TEE that it controls.

One way to solve this is for the Trusted Component Signer to run its own TAM that is only used to distribute the decryption key via the TEEP protocol, and the key file can be a dependency in the manifest of the encrypted TA. Thus, the TEEP Agent would look at the Trusted Component manifest, determine there is a dependency with a TAM URI of the Trusted Component Signer’s TAM. The Agent would then install the dependency, and then continue with the Trusted Component installation steps, including decrypting the TA binary with the relevant key.

9.9. REE Privacy

The TEEP architecture is applicable to cases where devices have a TEE that protects data and code from the REE administrator. In such cases, the TAM administrator, not the REE administrator, controls the TEE in the devices. As some examples:

* a cloud hoster may be the REE administrator where a customer administrator controls the TEE hosted in the cloud.

* a device manufacturer might control the TEE in a device purchased by a customer

The privacy risk is that data in the REE might be susceptible to disclosure to the TEE administrator. This risk is not introduced by the TEEP architecture, but is inherent in most uses of TEEs. This risk can be mitigated by making sure the REE administrator is aware of and explicitly chooses to have a TEE that is managed by another party. In the cloud hoster example, this choice is made by explicitly offering a service to customers to provide TEEs for them to administer. In the device manufacturer example, this choice is made by the customer choosing to buy a device made by a given manufacturer.
10.  IANA Considerations

This document does not require actions by IANA.

11.  Contributors

* Andrew Atyeo, Intercede (andrew.atyeo@intercede.com)
* Liu Dapeng, Alibaba Group (maxpassion@gmail.com)

12.  Acknowledgements

We would like to thank Nick Cook, Minho Yoo, Brian Witten, Tyler Kim, Alin Mutu, Juergen Schoenwaelder, Nicolae Paladi, Sorin Faibish, Ned Smith, Russ Housley, Jeremy O’Donoghue, and Anders Rundgren for their feedback.

13.  Informative References

[CC-Overview]

[CC-Technical-Analysis]

[GPTEE]

[GSMA]
[I-D.ietf-rats-architecture]

[I-D.ietf-suit-architecture]

[I-D.ietf-suit-manifest]

[I-D.ietf-teep-otrp-over-http]

[I-D.ietf-teep-protocol]

[OTRP]

[RFC4949]


Authors’ Addresses

Mingliang Pei
Broadcom
Email: mingliang.pei@broadcom.com

Hannes Tschofenig
Arm Limited
Email: hannes.tschofenig@arm.com

Dave Thaler
Microsoft
Email: dthaler@microsoft.com

David Wheeler
Amazon
Email: davewhee@amazon.com
HTTP Transport for Trusted Execution Environment Provisioning: Agent Initiated Communication
draft-ietf-teep-otrp-over-http-13

Abstract

The Trusted Execution Environment Provisioning (TEEP) Protocol is used to manage code and configuration data in a Trusted Execution Environment (TEE). This document specifies the HTTP transport for TEEP communication where a Trusted Application Manager (TAM) service is used to manage code and data in TEEs on devices that can initiate communication to the TAM. An implementation of this document can (if desired) run outside of any TEE, but interacts with a TEEP implementation that runs inside a TEE.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components
1. Introduction

A Trusted Execution Environment (TEE) is an environment that enforces that any code within that environment cannot be tampered with, and that any data used by such code cannot be read or tampered with by any code outside that environment. The Trusted Execution Environment Provisioning (TEEP) protocol is designed to provision authorized code and configuration into TEEs.

To be secure against malware, a TEEP implementation (referred to as a TEEP "Agent" on the client side, and a "Trusted Application Manager (TAM)" on the server side) SHOULD themselves run inside a TEE, although a TAM running outside a TEE is also supported. However, the
transport for TEEP, along with the underlying TCP/IP stack, does not necessarily run inside a TEE. This split allows the set of highly trusted code to be kept as small as possible, including allowing code (e.g., TCP/IP or QUIC [RFC9000]) that only sees encrypted messages, to be kept out of the TEE.

The TEEP specification [I-D.ietf-teep-protocol] (like its predecessors [I-D.ietf-teep-opentrustprotocol] and [GP-OTrP]) describes the behavior of TEEP Agents and TAMs, but does not specify the details of the transport. The purpose of this document is to provide such details. That is, a TEEP-over-HTTP (TEEP/HTTP) implementation delivers messages up to a TEEP implementation, and accepts messages from the TEEP implementation to be sent over a network. The TEEP-over-HTTP implementation can be implemented either outside a TEE (i.e., in a TEEP "Broker") or inside a TEE.

There are two topological scenarios in which TEEP could be deployed:

1. TAMs are reachable on the Internet, and Agents are on networks that might be behind a firewall or stateful NAT, so that communication must be initiated by an Agent. Thus, the Agent has an HTTP Client and the TAM has an HTTP Server.

2. Agents are reachable on the Internet, and TAMs are on networks that might be behind a firewall or stateful NAT, so that communication must be initiated by a TAM. Thus, the Agent has an HTTP Server and the TAM has an HTTP Client.

The remainder of this document focuses primarily on the first scenario as depicted in Figure 1, but some sections (Section 4 and Section 8) may apply to the second scenario as well. A fuller discussion of the second scenario may be handled by a separate document.

Figure 1: Agent Initiated Communication
This document specifies the middle layer (TEEP-over-HTTP), whereas the top layer (TEEP) is specified in [I-D.ietf-teep-protocol].

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document also uses various terms defined in [I-D.ietf-teep-architecture], including Trusted Execution Environment (TEE), Trusted Application (TA), Trusted Application Manager (TAM), TEEP Agent, TEEP Broker, and Rich Execution Environment (REE).

3. TEEP Broker

Section 6 of the TEEP architecture [I-D.ietf-teep-architecture] defines a TEEP "Broker" as being a component on the device, but outside the TEE, that facilitates communication with a TAM. That document further explains that the protocol layer at which the TEEP broker operates may vary by implementation, and it depicts several exemplary models. An implementation is free to choose any of these models, although model A is the one we will use in our examples.

Passing information from an REE component to a TEE component is typically spoken of as being passed "in" to the TEE, and information passed in the opposite direction is spoken of as being passed "out". In the protocol layering sense, information is typically spoken of as being passed "up" or "down" the stack. Since the layer at which information is passed in/out may vary by implementation, we will generally use "up" and "down" in this document.

3.1. Use of Abstract APIs

This document refers to various APIs between a TEEP implementation and a TEEP/HTTP implementation in the abstract, meaning the literal syntax and programming language are not specified, so that various concrete APIs can be designed (outside of the IETF) that are compliant.

Some TEE architectures (e.g., SGX) may support API calls both into and out of a TEE. In other TEE architectures, there may be no calls out from a TEE, but merely data returned from calls into a TEE. This document attempts to be agnostic as to the concrete API architecture for Broker/Agent communication. Since in model A, the Broker/Agent communication is done at the layer between the TEEP and TEEP/HTTP
implementations, and there may be some architectures that do not support calls out of the TEE (which would be downcalls from TEEP in model A), we will refer to passing information up to the TEEP implementation as API calls, but will simply refer to "passing data" back down from a TEEP implementation. A concrete API might pass data back via an API downcall or via data returned from an API upcall.

This document will also refer to passing "no" data back out of a TEEP implementation. In a concrete API, this might be implemented by not making any downcall, or by returning 0 bytes from an upcall, for example.

4. Use of HTTP as a Transport

This document uses HTTP [I-D.ietf-httpbis-semantics] as a transport. For the motivation behind the HTTP recommendations in this document, see the discussion of HTTP as a transport in [I-D.ietf-httpbis-bcp56bis].

Redirects MUST NOT be automatically followed. Cookies are not used.

Content is not intended to be treated as active by browsers and so HTTP responses with content SHOULD have the following header fields as explained in Section 4.13 of [I-D.ietf-httpbis-bcp56bis] (using the TEEP media type defined in [I-D.ietf-teep-protocol]):

- `Content-Type: application/teep+cbor`
- `X-Content-Type-Options: nosniff`
- `Content-Security-Policy: default-src 'none'`
- `Referrer-Policy: no-referrer`

Only the POST method is specified for TAM resources exposed over HTTP. A URI of such a resource is referred to as a "TAM URI". A TAM URI can be any HTTP(S) URI. The URI to use is configured in a TEEP Agent via an out-of-band mechanism, as discussed in the next section.

It is strongly RECOMMENDED that implementations use HTTPS. Although TEEP is protected end-to-end inside of HTTP, there is still value in using HTTPS for transport, since HTTPS can provide additional protections as discussed in Sections 4.4.2 and 6 of [I-D.ietf-httpbis-bcp56bis].

However, there may be constrained nodes where code space is an issue. [RFC7925] provides TLS profiles that can be used in many constrained nodes, but in rare cases the most constrained nodes might need to use HTTP without a TLS stack, relying on the end-to-end security provided by the TEEP protocol.
When HTTPS is used, clients MUST use the procedures detailed in Section 4.3.4 of [I-D.ietf-httpbis-semantics] to verify the authenticity of the server. See [BCP195] for additional TLS recommendations and [RFC7925] for TLS recommendations related to IoT devices.

5. TEEP/HTTP Client Behavior

5.1. Receiving a request to install a new Trusted Application

In some environments, an application installer can determine (e.g., from an application manifest) that the application being installed or updated has a dependency on a given Trusted Application (TA) being available in a given type of TEE. In such a case, it will notify a TEEP Broker, where the notification will contain the following:

* A unique identifier of the TA

* Optionally, any metadata to provide to the TEEP Agent. This might include a TAM URI provided in the application manifest, for example.

* Optionally, any requirements that may affect the choice of TEE, if multiple are available to the TEEP Broker.

When a TEEP Broker receives such a notification, it first identifies in an implementation-dependent way which TEE (if any) is most appropriate based on the constraints expressed. If there is only one TEE, the choice is obvious. Otherwise, the choice might be based on factors such as capabilities of available TEE(s) compared with TEE requirements in the notification. Once the TEEP Broker picks a TEE, it passes the notification to the TEEP/HTTP Client for that TEE.

The TEEP/HTTP Client then informs the TEEP Agent in that TEE by invoking an appropriate "RequestTA" API that identifies the TA needed and any other associated metadata. The TEEP/HTTP Client need not know whether the TEE already has such a TA installed or whether it is up to date.

The TEEP Agent will either (a) pass no data back, (b) pass back a TAM URI to connect to, or (c) pass back a message buffer and TAM URI to send it to. The TAM URI passed back may or may not be the same as the TAM URI, if any, provided by the TEEP/HTTP Client, depending on the TEEP Agent’s configuration. If they differ, the TEEP/HTTP Client MUST use the TAM URI passed back.
5.1.1. Session Creation

If no data is passed back, the TEEP/HTTP Client simply informs its caller (e.g., the application installer) of success.

If the TEEP Agent passes back a TAM URI with no message buffer, the TEEP/HTTP Client attempts to create session state, then sends an HTTP(S) POST to the TAM URI with an Accept header field with the TEEP media type specified in [I-D.ietf-teep-protocol], and an empty body. The HTTP request is then associated with the TEEP/HTTP Client’s session state.

If the TEEP Agent instead passes back a TAM URI with a message buffer, the TEEP/HTTP Client attempts to create session state and handles the message buffer as specified in Section 5.3.

Session state consists of:

* Any context (e.g., a handle) that identifies the API session with the TEEP Agent.

* Any context that identifies an HTTP request, if one is outstanding. Initially, none exists.

5.2. Receiving a notification that a Trusted Application is no longer needed

In some environments, an application installer can determine (e.g., from an application manifest) that a given Trusted Application is no longer needed, such as if the application that previously depended on the TA is uninstalled or updated in a way that removes the dependency. In such a case, it will notify a TEEP Broker, where the notification will contain the following:

* A unique identifier of the TA

* Optionally, any metadata to provide to the TEEP Agent. This might include a TAM URI provided in the original application manifest, for example.

* Optionally, any requirements that may affect the choice of TEE, if multiple are available to the TEEP Broker.

When a TEEP Broker receives such a notification, it first identifies in an implementation-dependent way which TEE (if any) is believed to contain the TA that is no longer needed, similar to the process in Section 5.1. Once the TEEP Broker picks a TEE, it passes the notification to the TEEP/HTTP Client for that TEE.
The TEEP/HTTP Client then informs the TEEP Agent in that TEE by invoking an appropriate "UnrequestTA" API that identifies the unneeded TA. The TEEP/HTTP Client need not know whether the TEE actually has the TA installed.

The TEEP Agent will either (a) pass no data back, (b) pass back a TAM URI to connect to, or (c) pass back a message buffer and TAM URI to send it to. The TAM URI passed back may or may not be the same as the TAM URI, if any, provided by the TEEP/HTTP Client, depending on the TEEP Agent’s configuration. If they differ, the TEEP/HTTP Client MUST use the TAM URI passed back.

Processing then continues as in Section 5.1.1.

5.3. Getting a TAM URI and message buffer back from a TEEP Agent

When a TEEP Agent passes a TAM URI and optionally a message buffer to a TEEP/HTTP Client, the TEEP/HTTP Client MUST do the following, using the TEEP/HTTP Client’s session state associated with its API call to the TEEP Agent.

The TEEP/HTTP Client sends an HTTP POST request to the TAM URI with Accept and Content-Type header fields with the TEEP media type, and a body containing the TEEP message buffer (if any) provided by the TEEP Agent. The HTTP request is then associated with the TEEP/HTTP Client’s session state.

5.4. Receiving an HTTP response

When an HTTP response is received in response to a request associated with a given session state, the TEEP/HTTP Client MUST do the following.

If the HTTP response body is empty, the TEEP/HTTP Client’s task is complete, and it can delete its session state, and its task is done.

If instead the HTTP response body is not empty, the TEEP/HTTP Client passes (e.g., using the "ProcessTeepMessage" API as mentioned in Section 6.2.1 of [I-D.ietf-teep-architecture]) the response body up to the TEEP Agent associated with the session. The TEEP Agent will then either pass no data back, or pass back a message buffer.

If no data is passed back, the TEEP/HTTP Client’s task is complete, and it can delete its session state, and inform its caller (e.g., the application installer) of success.

If instead the TEEP Agent passes back a message buffer, the TEEP/HTTP Client handles the message buffer as specified in Section 5.3.
5.5. Handling checks for policy changes

An implementation MUST provide a way to periodically check for TAM policy changes, such as a Trusted Application needing to be deleted from a TEE because it is no longer permitted, or needing to be updated to a later version. This can be done in any implementation-specific manner, such as any of the following or a combination thereof:

A) The TEEP/HTTP Client might call up to the TEEP Agent at an interval previously specified by the TEEP Agent. This approach requires that the TEEP/HTTP Client be capable of running a periodic timer.

B) The TEEP/HTTP Client might be informed when an existing TA is invoked, and call up to the TEEP Agent if more time has passed than was previously specified by the TEEP Agent. This approach allows the device to go to sleep for a potentially long period of time.

C) The TEEP/HTTP Client might be informed when any attestation attempt determines that the device is out of compliance, and call up to the TEEP Agent to remediate.

The TEEP/HTTP Client informs the TEEP Agent by invoking an appropriate "RequestPolicyCheck" API. The TEEP Agent will either (a) pass no data back, (b) pass back a TAM URI to connect to, or (c) pass back a message buffer and TAM URI to send it to. Processing then continues as specified in Section 5.1.1.

The TEEP Agent might need to talk to multiple TAMs, however, as shown in Figure 1 of [I-D.ietf-teep-architecture]. To accomplish this, the TEEP/HTTP Client keeps invoking the "RequestPolicyCheck" API until the TEEP Agent passes no data back, so that the TEEP Agent can return each TAM URI in response to a separate API call.

5.6. Error handling

If any local error occurs where the TEEP/HTTP Client cannot get a message buffer (empty or not) back from the TEEP Agent, the TEEP/HTTP Client deletes its session state, and informs its caller (if any, e.g., the application installer) of a failure.

If any HTTP request results in an HTTP error response or a lower layer error (e.g., network unreachable), the TEEP/HTTP Client calls the TEEP Agent’s "ProcessError" API, and then deletes its session state and informs its caller of a failure.
6. TEEP/HTTP Server Behavior

6.1. Receiving an HTTP POST request

If the TAM does not receive the appropriate Content-Type header field value, the TAM SHOULD fail the request, returning a 415 Unsupported Media Type response. Similarly, if an appropriate Accept header field is not present, the TAM SHOULD fail the request with an appropriate error response. (This is for consistency with common implementation practice to allow the HTTP server to choose a default error response, since in some implementations the choice is done at the HTTP layer rather than the layer at which TEEP-over-HTTP would be implemented.) Otherwise, processing continues as follows.

When an HTTP POST request is received with an empty body, this indicates a request for a new TEEP session, and the TEEP/HTTP Server invokes the TAM’s "ProcessConnect" API. The TAM will then pass back a message buffer.

When an HTTP POST request is received with a non-empty body, this indicates a message on an existing TEEP session, and the TEEP/HTTP Server passes the request body to the TAM (e.g., using the "ProcessTeepMessage" API mentioned in [I-D.ietf-teep-architecture]). The TAM will then pass back a (possibly empty) message buffer.

6.2. Getting an empty buffer back from the TAM

If the TAM passes back an empty buffer, the TEEP/HTTP Server sends a successful (2xx) response with no body. It SHOULD be status 204 (No Content).

6.3. Getting a message buffer from the TAM

If the TAM passes back a non-empty buffer, the TEEP/HTTP Server generates a successful (2xx) response with a Content-Type header field with the TEEP media type, and with the message buffer as the body.

6.4. Error handling

If any error occurs where the TEEP/HTTP Server cannot get a message buffer (empty or not) back from the TAM, the TEEP/HTTP Server generates an appropriate HTTP 5xx error response.

7. Sample message flow

The following shows a sample TEEP message flow that uses application/teep+cbor as the Content-Type.
1. An application installer determines (e.g., from an application manifest) that the application has a dependency on TA "X", and passes this notification to the TEEP Broker. The TEEP Broker picks a TEE (e.g., the only one available) based on this notification, and passes the information to the TEEP/HTTP Client for that TEE.

2. The TEEP/HTTP Client calls the TEEP Agent’s "RequestTA" API, passing TA Needed = X.

3. The TEEP Agent finds that no such TA is already installed, but that it can be obtained from a given TAM. The TEEP Agent passes back the TAM URI (e.g., "https://example.com/tam") to the TEEP/HTTP Client.

4. The TEEP/HTTP Client sends an HTTP POST request to the TAM URI:

   POST /tam HTTP/1.1
   Host: example.com
   Accept: application/teep+cbor
   Content-Length: 0
   User-Agent: Foo/1.0

   where the TEEP/HTTP Client fills in an implementation-specific value in the User-Agent header field.

5. On the TAM side, the TEEP/HTTP Server receives the HTTP POST request, and calls the TAM’s "ProcessConnect" API.

6. The TAM generates a TEEP message (where typically QueryRequest is the first message) and passes it to the TEEP/HTTP Server.

7. The TEEP/HTTP Server sends an HTTP successful response with the TEEP message in the body:

   HTTP/1.1 200 OK
   Content-Type: application/teep+cbor
   Content-Length: [length of TEEP message here]
   Server: Bar/2.2
   X-Content-Type-Options: nosniff
   Content-Security-Policy: default-src 'none'
   Referrer-Policy: no-referrer

   [TEEP message here]

   where the TEEP/HTTP Server fills in an implementation-specific value in the Server header field.
8. Back on the TEEP Agent side, the TEEP/HTTP Client gets the HTTP response, extracts the TEEP message and pass it up to the TEEP Agent.

9. The TEEP Agent processes the TEEP message, and generates a TEEP response (e.g., QueryResponse) which it passes back to the TEEP/HTTP Client.

10. The TEEP/HTTP Client gets the TEEP message buffer and sends an HTTP POST request to the TAM URI, with the TEEP message in the body:

    POST /tam HTTP/1.1
    Host: example.com
    Accept: application/teep+cbor
    Content-Type: application/teep+cbor
    Content-Length: [length of TEEP message here]
    User-Agent: Foo/1.0

    [TEEP message here]

11. The TEEP/HTTP Server receives the HTTP POST request, and passes the payload up to the TAM.

12. Steps 6-11 are then repeated until the TAM passes no data back to the TEEP/HTTP Server in step 6.

13. The TEEP/HTTP Server sends an HTTP successful response with no body:

    HTTP/1.1 204 No Content
    Server: Bar/2.2

14. The TEEP/HTTP Client deletes its session state.

8. Security Considerations

   Section 4 discussed security recommendations for HTTPS transport of TEEP messages. See Section 6 of [I-D.ietf-httpbis-bcp56bis] for additional discussion of HTTP(S) security considerations. See section 9 of [I-D.ietf-teep-architecture] for security considerations specific to the use of TEEP.

9. IANA Considerations

   This document has no actions for IANA.

10. References
10.1. Normative References


10.2. Informative References


Author’s Address

Dave Thaler
Microsoft

Email: dthaler@microsoft.com
Trusted Execution Environment Provisioning (TEEP) Protocol
draft-ietf-teep-protocol-08

Abstract

This document specifies a protocol that installs, updates, and deletes Trusted Components in a device with a Trusted Execution Environment (TEE). This specification defines an interoperable protocol for managing the lifecycle of Trusted Components.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document.
Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction .................................................. 3
2. Terminology .................................................... 4
3. Message Overview ............................................... 4
4. Detailed Messages Specification .............................. 6
   4.1. Creating and Validating TEEP Messages .................. 6
   4.1.1. Creating a TEEP message ............................... 6
   4.1.2. Validating a TEEP Message ............................. 6
   4.2. QueryRequest Message .................................... 7
   4.3. QueryResponse Message ................................... 9
   4.3.1. Evidence and Attestation Results .................... 12
   4.4. Update Message .......................................... 13
   4.4.1. Example 1: Having one SUIT Manifest pointing to a URI of a Trusted Component Binary ..................... 15
   4.4.2. Example 2: Having a SUIT Manifest include the Trusted Component Binary ................................. 17
   4.4.3. Example 3: Supplying Personalization Data for the Trusted Component Binary ............................. 18
   4.4.4. Example 4: Unlinking Trusted Component ............... 20
   4.5. Success Message ......................................... 21
   4.6. Error Message ........................................... 22
5. EAT Profile .................................................... 25
6. Mapping of TEEP Message Parameters to CBOR Labels ........ 27
7. Behavior Specification .......................................... 29
   7.1. TAM Behavior ............................................ 29
   7.2. TEEP Agent Behavior ..................................... 30
8. Ciphersuites .................................................... 31
9. Freshness Mechanisms .......................................... 32
10. Security Considerations ....................................... 33
11. IANA Considerations .......................................... 35
   11.1. Media Type Registration ................................ 35
   11.2. Freshness Mechanism Registry ........................... 35
12. References ..................................................... 36
   12.1. Normative References ................................... 36
   12.2. Informative References ................................ 38
A. Contributors .................................................... 39
B. Acknowledgements ............................................. 39
C. Complete CDDL .................................................. 39
D. Examples of Diagnostic Notation and Binary Representation .. 43
   D.1. QueryRequest Message .................................... 43
      D.1.1. CBOR Diagnostic Notation ............................ 43
1. Introduction

The Trusted Execution Environment (TEE) concept has been designed to separate a regular operating system, also referred as a Rich Execution Environment (REE), from security-sensitive applications. In a TEE ecosystem, device vendors may use different operating systems in the REE and may use different types of TEEs. When Trusted Component Developers or Device Administrators use Trusted Application
Managers (TAMs) to install, update, and delete Trusted Applications and their dependencies on a wide range of devices with potentially different TEEs then an interoperability need arises.

This document specifies the protocol for communicating between a TAM and a TEEP Agent.

The Trusted Execution Environment Provisioning (TEEP) architecture document [I-D.ietf-teep-architecture] provides design guidance and introduces the necessary terminology.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification re-uses the terminology defined in [I-D.ietf-teep-architecture].

As explained in Section 4.4 of that document, the TEEP protocol treats each Trusted Application (TA), any dependencies the TA has, and personalization data as separate components that are expressed in SUIT manifests, and a SUIT manifest might contain or reference multiple binaries (see [I-D.ietf-suit-manifest] for more details).

As such, the term Trusted Component (TC) in this document refers to a set of binaries expressed in a SUIT manifest, to be installed in a TEE. Note that a Trusted Component may include one or more TAs and/or configuration data and keys needed by a TA to operate correctly.

Each Trusted Component is uniquely identified by a SUIT Component Identifier (see [I-D.ietf-suit-manifest] Section 8.7.2.2).

3. Message Overview

The TEEP protocol consists of messages exchanged between a TAM and a TEEP Agent. The messages are encoded in CBOR and designed to provide end-to-end security. TEEP protocol messages are signed by the endpoints, i.e., the TAM and the TEEP Agent, but Trusted Applications may also be encrypted and signed by a Trusted Component Developer or Device Administrator. The TEEP protocol not only uses CBOR but also the respective security wrapper, namely COSE [RFC8152]. Furthermore, for software updates the SUIT manifest format [I-D.ietf-suit-manifest] is used, and for attestation the Entity Attestation Token (EAT) [I-D.ietf-rats-eat] format is supported.
although other attestation formats are also permitted.

This specification defines five messages: QueryRequest, QueryResponse, Update, Success, and Error.

A TAM queries a device’s current state with a QueryRequest message. A TEEP Agent will, after authenticating and authorizing the request, report attestation information, list all Trusted Components, and provide information about supported algorithms and extensions in a QueryResponse message. An error message is returned if the request could not be processed. A TAM will process the QueryResponse message and determine whether to initiate subsequent message exchanges to install, update, or delete Trusted Applications.

```
+------------+           +-------------+
| TAM        |           | TEEP Agent   |
+------------+           +-------------+
     QueryRequest -------->       QueryResponse
                                     <------       or
                                     Error
```

With the Update message a TAM can instruct a TEEP Agent to install and/or delete one or more Trusted Components. The TEEP Agent will process the message, determine whether the TAM is authorized and whether the Trusted Component has been signed by an authorized Trusted Component Signer. A Success message is returned when the operation has been completed successfully, or an Error message otherwise.

```
+------------+           +-------------+
| TAM        |           | TEEP Agent   |
+------------+           +-------------+
     Update    ---->       Success
                                     <----       or
                                     Error
```
4. Detailed Messages Specification

TEEP messages are protected by the COSE_Sign1 structure. The TEEP protocol messages are described in CDDL format [RFC8610] below.

\{
    tEEP-message => (query-request /
        query-response /
        update /
        tEEP-success /
        tEEP-error ),
\}

4.1. Creating and Validating TEEP Messages

4.1.1. Creating a TEEP message

To create a TEEP message, the following steps are performed.

1. Create a TEEP message according to the description below and populate it with the respective content. TEEP messages sent by TAMs (QueryRequest and Update) can include a "token". The TAM can decide, in any implementation-specific way, whether to include a token in a message. The first usage of a token generated by a TAM MUST be randomly created. Subsequent token values MUST be different for each subsequent message created by a TAM.

2. Create a COSE Header containing the desired set of Header Parameters. The COSE Header MUST be valid per the [RFC8152] specification.

3. Create a COSE_Sign1 object using the TEEP message as the COSE_Sign1 Payload; all steps specified in [RFC8152] for creating a COSE_Sign1 object MUST be followed.

4.1.2. Validating a TEEP Message

When TEEP message is received (see the ProcessTeepMessage conceptual API defined in [I-D.ietf-teep-architecture] section 6.2.1), the following validation steps are performed. If any of the listed steps fail, then the TEEP message MUST be rejected.

1. Verify that the received message is a valid CBOR object.

2. Verify that the message contains a COSE_Sign1 structure.
3. Verify that the resulting COSE Header includes only parameters and values whose syntax and semantics are both understood and supported or that are specified as being ignored when not understood.

4. Follow the steps specified in Section 4 of [RFC8152] ("Signing Objects") for validating a COSE_Sign1 object. The COSE_Sign1 payload is the content of the TEEP message.

5. Verify that the TEEP message is a valid CBOR map and verify the fields of the TEEP message according to this specification.

4.2. QueryRequest Message

A QueryRequest message is used by the TAM to learn information from the TEEP Agent, such as the features supported by the TEEP Agent, including ciphersuites and protocol versions. Additionally, the TAM can selectively request data items from the TEEP Agent via the request parameter. Currently, the following features are supported:

* Request for attestation information,
* Listing supported extensions,
* Querying installed Trusted Components, and
* Listing supported SUIT commands.

Like other TEEP messages, the QueryRequest message is signed, and the relevant CDDL snippet is shown below. The complete CDDL structure is shown in Appendix C.

```plaintext
query-request = [ 
  type: TEEP-TYPE-query-request, 
  options: { 
    ? token => bstr .size (8..64), 
    ? supported-cipher-suites => [ + suite ], 
    ? supported-freshness-mechanisms => [ + freshness-mechanism ], 
    ? challenge => bstr .size (8..512), 
    ? versions => [ + version ], 
    * $$query-request-extensions 
    * $$teep-option-extensions 
  }, 
  data-item-requested: data-item-requested 
]
```

The message has the following fields:
type
The value of (1) corresponds to a QueryRequest message sent from
the TAM to the TEEP Agent.

token
The value in the token parameter is used to match responses to
requests. This is particularly useful when a TAM issues multiple
concurrent requests to a TEEP Agent. The token MUST be present if
and only if the attestation bit is clear in the data-item-
requested value. The size of the token is at least 8 bytes (64
bits) and maximum of 64 bytes, which is the same as in an EAT
Nonce Claim (see [I-D.ietf-rats-eat] Section 3.3). The first
usage of a token generated by a TAM MUST be randomly created.
Subsequent token values MUST be different for each request message
to distinguish the correct response from multiple requests. The
token value MUST NOT be used for other purposes, such as to identify
the devices and/or a device to identify TAMs or Trusted
Components. The TAM SHOULD set an expiration time for each token
and MUST ignore any messages with expired tokens. The TAM MUST
expire the token value after receiving the first response
containing the token value and ignore any subsequent messages that
have the same token value.

data-item-requested
The data-item-requested parameter indicates what information the
TAM requests from the TEEP Agent in the form of a bitmap. Each
value in the bitmap corresponds to an IANA registered information
element. This specification defines the following initial set of
information elements:

attestation (1) With this value the TAM requests the TEEP Agent
to return attestation evidence (e.g., an EAT) in the response.

trusted-components (2) With this value the TAM queries the TEEP
Agent for all installed Trusted Components.

extensions (4) With this value the TAM queries the TEEP Agent for
supported capabilities and extensions, which allows a TAM to
discover the capabilities of a TEEP Agent implementation.

Further values may be added in the future via IANA registration.
supported-cipher-suites
The supported-cipher-suites parameter lists the ciphersuites
supported by the TAM. If this parameter is not present, it is to
be treated the same as if it contained all ciphersuites defined in
this document that are listed as "MUST". Details about the
ciphersuite encoding can be found in Section 8.
supported-freshness-mechanisms

The supported-freshness-mechanisms parameter lists the freshness mechanism(s) supported by the TAM. Details about the encoding can be found in Section 9. If this parameter is absent, it means only the nonce mechanism is supported.

challenge

The challenge field is an optional parameter used for ensuring the freshness of the attestation evidence returned with a QueryResponse message. It MUST be absent if the attestation bit is clear (since the token is used instead in that case). When a challenge is provided in the QueryRequest and an EAT is returned with a QueryResponse message then the challenge contained in this request MUST be used to generate the EAT, such as by copying the challenge into the nonce claim found in the EAT if using the Nonce freshness mechanism. For more details see Section 9. If any format other than EAT is used, it is up to that format to define the use of the challenge field.

versions

The versions parameter enumerates the TEEP protocol version(s) supported by the TAM. A value of 0 refers to the current version of the TEEP protocol. If this field is not present, it is to be treated the same as if it contained only version 0.

4.3. QueryResponse Message

The QueryResponse message is the successful response by the TEEP Agent after receiving a QueryRequest message. As discussed in Section 7.2, it can also be sent unsolicited if the contents of the QueryRequest are already known and do not vary per message.

Like other TEEP messages, the QueryResponse message is signed, and the relevant CDDL snippet is shown below. The complete CDDL structure is shown in Appendix C.
query-response = [
  type: TEEP-TYPE-query-response,
  options: {
    ? token => bstr .size (8..64),
    ? selected-cipher-suite => suite,
    ? selected-version => version,
    ? evidence-format => text,
    ? evidence => bstr,
    ? tc-list => [ + tc-info ],
    ? requested-tc-list => [ + requested-tc-info ],
    ? unneeded-tc-list => [ + SUIT_Component_Identifier ],
    ? ext-list => [ + ext-info ],
    * $$query-response-extensions,
    * $$teep-option-extensions
  }
]

tc-info = {
  component-id => SUIT_Component_Identifier,
  ? tc-manifest-sequence-number => .within uint .size 8
}

requested-tc-info = {
  component-id => SUIT_Component_Identifier,
  ? tc-manifest-sequence-number => .within uint .size 8
  ? have-binary => bool
}

The QueryResponse message has the following fields:

type
  The value of (2) corresponds to a QueryResponse message sent from
  the TEEP Agent to the TAM.

token
  The value in the token parameter is used to match responses to
  requests. The value MUST correspond to the value received with
  the QueryRequest message if one was present, and MUST be absent if
  no token was present in the QueryRequest.

selected-cipher-suite
  The selected-cipher-suite parameter indicates the selected
  ciphersuite. Details about the ciphersuite encoding can be found
  in Section 8.
selected-version

The selected-version parameter indicates the TEEP protocol version selected by the TEEP Agent. The absence of this parameter indicates the same as if it was present with a value of 0.

evidence-format

The evidence-format parameter indicates the IANA Media Type of the attestation evidence contained in the evidence parameter. It MUST be present if the evidence parameter is present and the format is not an EAT.

evidence

The evidence parameter contains the attestation evidence. This parameter MUST be present if the QueryResponse is sent in response to a QueryRequest with the attestation bit set. If the evidence-format parameter is absent, the attestation evidence contained in this parameter MUST be an Entity Attestation Token following the encoding defined in [I-D.ietf-rats-eat]. See Section 4.3.1 for further discussion.

tc-list

The tc-list parameter enumerates the Trusted Components installed on the device in the form of tc-info objects. This parameter MUST be present if the QueryResponse is sent in response to a QueryRequest with the trusted-components bit set.

requested-tc-list

The requested-tc-list parameter enumerates the Trusted Components that are not currently installed in the TEE, but which are requested to be installed, for example by an installer of an Untrusted Application that has a TA as a dependency, or by a Trusted Application that has another Trusted Component as a dependency. Requested Trusted Components are expressed in the form of requested-tc-info objects. A TEEP Agent can get this information from the RequestTA conceptual API defined in [I-D.ietf-teep-architecture] section 6.2.1.

unneeded-tc-list

The unneeded-tc-list parameter enumerates the Trusted Components that are currently installed in the TEE, but which are no longer needed by any other application. The TAM can use this information in determining whether a Trusted Component can be deleted. Each unneeded Trusted Component is identified by its SUIT Component Identifier. A TEEP Agent can get this information from the UnrequestTA conceptual API defined in [I-D.ietf-teep-architecture] section 6.2.1.
The `ext-list` parameter lists the supported extensions. This document does not define any extensions. This parameter MUST be present if the QueryResponse is sent in response to a QueryRequest with the extensions bit set.

The `tc-info` object has the following fields:

- **component-id**
  
  A SUIT Component Identifier.

- **tc-manifest-sequence-number**
  
  The suit-manifest-sequence-number value from the SUIT manifest for the Trusted Component, if a SUIT manifest was used.

The requested-tc-info message has the following fields:

- **component-id**
  
  A SUIT Component Identifier.

- **tc-manifest-sequence-number**
  
  The minimum suit-manifest-sequence-number value from a SUIT manifest for the Trusted Component. If not present, indicates that any sequence number will do.

- **have-binary**
  
  If present with a value of true, indicates that the TEEP agent already has the Trusted Component binary and only needs an Update message with a SUIT manifest that authorizes installing it. If have-binary is true, the tc-manifest-sequence-number field MUST be present.

### 4.3.1. Evidence and Attestation Results

Section 7 of [I-D.ietf-teep-architecture] lists information that may appear in evidence depending on the circumstance. However, the evidence is opaque to the TEEP protocol and there are no formal requirements on the contents of evidence.

TAMs however consume Attestation Results and do need enough information therein to make decisions on how to remediate a TEE that is out of compliance, or update a TEE that is requesting an authorized change. To do so, the information in Section 7 of [I-D.ietf-teep-architecture] is often required depending on the policy. When an Entity Attestation Token is used, the following claims can be used to meet those requirements:
### Table 1

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Claim</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device unique identifier</td>
<td>ueid</td>
<td>[I-D.ietf-rats-eat] section 3.4</td>
</tr>
<tr>
<td>Vendor of the device</td>
<td>oemid</td>
<td>[I-D.ietf-rats-eat] section 3.6</td>
</tr>
<tr>
<td>Class of the device</td>
<td>class-identifier</td>
<td>[I-D.birkholz-rats-suit-claims] section 3.1.2</td>
</tr>
<tr>
<td>TEE hardware type</td>
<td>chip-version</td>
<td>[I-D.ietf-rats-eat] section 3.7</td>
</tr>
<tr>
<td>TEE hardware version</td>
<td>chip-version</td>
<td>[I-D.ietf-rats-eat] section 3.7</td>
</tr>
<tr>
<td>TEE firmware type</td>
<td>sw-name</td>
<td>[I-D.ietf-rats-eat] section 3.9</td>
</tr>
<tr>
<td>TEE firmware version</td>
<td>sw-version</td>
<td>[I-D.ietf-rats-eat] section 3.10</td>
</tr>
<tr>
<td>Freshness proof</td>
<td>nonce</td>
<td>[I-D.ietf-rats-eat] section 3.3</td>
</tr>
</tbody>
</table>

#### 4.4. Update Message

The Update message is used by the TAM to install and/or delete one or more Trusted Components via the TEEP Agent.

Like other TEEP messages, the Update message is signed, and the relevant CDDL snippet is shown below. The complete CDDL structure is shown in Appendix C.
update = [
  type: TEEP-TYPE-update,
  options: {
    ? token => bstr .size (8..64),
    ? manifest-list => [ + bstr .cbor SUIT_Envelope ],
    * $$update-extensions,
    * $$teep-option-extensions
  }
]

The Update message has the following fields:

type
The value of (3) corresponds to an Update message sent from the TAM to the TEEP Agent. In case of successful processing, a Success message is returned by the TEEP Agent. In case of an error, an Error message is returned. Note that the Update message is used for initial Trusted Component installation as well as for updates and deletes.

token
The value in the token field is used to match responses to requests.

manifest-list
The manifest-list field is used to convey one or multiple SUIT manifests to install. A manifest is a bundle of metadata about a Trusted Component, such as where to find the code, the devices to which it applies, and cryptographic information protecting the manifest. The manifest may also convey personalization data. Trusted Component binaries and personalization data can be signed and encrypted by the same Trusted Component Signer. Other combinations are, however, possible as well. For example, it is also possible for the TAM to sign and encrypt the personalization data and to let the Trusted Component Developer sign and/or encrypt the Trusted Component binary.

Note that an Update message carrying one or more SUIT manifests will inherently involve multiple signatures, one by the TAM in the TEEP message and one from a Trusted Component Signer inside each manifest. This is intentional as they are for different purposes.

The TAM is what authorizes apps to be installed, updated, and deleted on a given TEE and so the TEEP signature is checked by the TEEP Agent at protocol message processing time. (This same TEEP security wrapper is also used on messages like QueryRequest so that Agents only send potentially sensitive data such as evidence to trusted TAMs.)
The Trusted Component signer on the other hand is what authorizes the Trusted Component to actually run, so the manifest signature could be checked at install time or load (or run) time or both, and this checking is done by the TEE independent of whether TEEP is used or some other update mechanism. See section 5 of [I-D.ietf-teep-architecture] for further discussion.

The Update Message has a SUIT_Envelope containing SUIT manifests. Following are some examples of using SUIT manifests in the Update Message.

4.4.1. Example 1: Having one SUIT Manifest pointing to a URI of a Trusted Component Binary

In this example, a SUIT Manifest has a URI pointing to a Trusted Component Binary.

A Trusted Component Developer creates a new Trusted Component Binary and hosts it at a Trusted Component Developer’s URI. Then the Trusted Component Developer generates an associated SUIT manifest with the filename "tc-uuid.suit" that contains the URI. The filename "tc-uuid.suit" is used in Example 3 later.

The TAM receives the latest SUIT manifest from the Trusted Component Developer, and the URI it contains will not be changeable by the TAM since the SUIT manifest is signed by the Trusted Component Developer.

Pros:

* The Trusted Component Developer can ensure that the intact Trusted Component Binary is downloaded by devices
* The TAM does not have to send large Update messages containing the Trusted Component Binary

Cons:

* The Trusted Component Developer must host the Trusted Component Binary server
* The device must fetch the Trusted Component Binary in another connection after receiving an Update message
Figure 1: URI of the Trusted Component Binary

For the full SUIT Manifest example binary, see Appendix "Example 1: SUIT Manifest pointing to URI of the Trusted Component Binary".
4.4.2. Example 2: Having a SUIT Manifest include the Trusted Component Binary

In this example, the SUIT manifest contains the entire Trusted Component Binary using the integrated-payload (see [I-D.ietf-suit-manifest] Section 7.6).

A Trusted Component Developer delegates to the TAM the task of delivering the Trusted Component Binary in the SUIT manifest. The Trusted Component Developer creates a SUIT manifest and embeds the Trusted Component Binary, which is referenced in the URI parameter with identifier "#tc". The Trusted Component Developer provides the SUIT manifest to the TAM.

The TAM serves the SUIT manifest containing the Trusted Component Binary to the device in an Update message.

Pros:

* The device can obtain the Trusted Component Binary and its SUIT manifest together in one Update message

* The Trusted Component Developer does not have to host a server to deliver the Trusted Component Binary directly to devices

Cons:

* The TAM must host the Trusted Component Binary itself, rather than delegating such storage to the Trusted Component Developer

* The TAM must deliver Trusted Component Binaries in Update messages, which result in increased Update message size
4.4.3. Example 3: Supplying Personalization Data for the Trusted Component Binary

In this example, Personalization Data is associated with the Trusted Component Binary "tc-uuid.suit" from Example 1.

The Trusted Component Developer places Personalization Data in a file named "config.json" and hosts it on an HTTPS server. The Trusted Component Developer then creates a SUIT manifest with the URI, specifying which Trusted Component Binary it correlates to in the parameter 'dependency-resolution', and signs the SUIT manifest.

The TAM delivers the SUIT manifest of the Personalization Data which depends on the Trusted Component Binary from Example 1.
Update ---->

```
Teep-protocol(TAM) ===============

Teep-Message({
    Teep-type-update,
    options: {
        manifest-list: [
            suit-manifest(TC Developer) ===============
            suit-Envelope(
                manifest: {
                    common: {
                        dependencies: [(
                            digest-of-tc.suit)
                        ],
                    }
                }
            dependency-resolution: {
                set-parameter: {
                    uri: "https://example.org/tc-uuid.suit"
                }
            }
            install: {
                set-parameter: {
                    uri: "https://example.org/config.json"
                }
            }
        }
    }
})
```

and then,

```
Teep-Agent ---->
```

Tschofenig, et al.     Expires 8 September 2022
fetch "https://example.org/config.json"

```
+=======config.json========+
| 7B 22 75 73 65 72 22 ... |
+==========================+
```

Figure 3: Personalization Data

For the full SUIT Manifest example binary, see Appendix "Example 3: Supplying Personalization Data for Trusted Component Binary".

4.4.4. Example 4: Unlinking Trusted Component

This subsection shows an example deleting the Trusted Component Binary in the TEEP Device.

A Trusted Component Developer can also generate SUIT Manifest which unlinks the installed Trusted Component. The TAM deliver it when the TAM want to uninstall the component.

The directive-unlink (see [I-D.moran-suit-trust-domains] Section-6.5.4) is located in the manifest to delete the Trusted Component. Note that in case other Trusted Components depend on it, i.e. the reference count is not zero, the TEEP Device SHOULD NOT delete it immediately.
Figure 4: Unlink Trusted Component example (summary)

For the full SUIT Manifest example binary, see Appendix E. SUIT Example 4 (Appendix "E.4. Example 4: Unlink a Trusted Component")

4.5. Success Message

The Success message is used by the TEEP Agent to return a success in response to an Update message.

Like other TEEP messages, the Success message is signed, and the relevant CDDL snippet is shown below. The complete CDDL structure is shown in Appendix C.
teep-success = [
  type: TEEP-TYPE-teep-success,
  options: {
    ? token => bstr .size (8..64),
    ? msg => text .size (1..128),
    ? suit-reports => [ + suit-report ],
    * $$teep-success-extensions,
    * $$teep-option-extensions
  }
]

The Success message has the following fields:

type
The value of (5) corresponds to a Success message sent from the TEEP Agent to the TAM.

token
The value in the token parameter is used to match responses to requests. It MUST match the value of the token parameter in the Update message the Success is in response to, if one was present. If none was present, the token MUST be absent in the Success message.

msg
The msg parameter contains optional diagnostics information encoded in UTF-8 [RFC3629] using Net-Unicode form [RFC5198] with max 128 bytes returned by the TEEP Agent.

suit-reports
If present, the suit-reports parameter contains a set of SUIT Reports as defined in Section 4 of [I-D.moran-suit-report]. If a token parameter was present in the Update message the Success message is in response to, the suit-report-nonce field MUST be present in the SUIT Report with a value matching the token parameter in the Update message.

4.6. Error Message

The Error message is used by the TEEP Agent to return an error in response to an Update message.

Like other TEEP messages, the Error message is signed, and the relevant CDDL snippet is shown below. The complete CDDL structure is shown in Appendix C.
teep-error = [
  type: TEEP-TYPE-teep-error,
  options: {
    ? token => bstr .size (8..64),
    ? err-msg => text .size (1..128),
    ? supported-cipher-suites => [ + suite ],
    ? supported-freshness-mechanisms => [ + freshness-mechanism ],
    ? versions => [ + version ],
    ? suit-reports => [ + suit-report ],
    * $$teep-error-extensions,
    * $$teep-option-extensions
  },
  err-code: uint (0..23)
]

The Error message has the following fields:

type
The value of (6) corresponds to an Error message sent from the TEEP Agent to the TAM.

token
The value in the token parameter is used to match responses to requests. It MUST match the value of the token parameter in the Update message the Success is in response to, if one was present. If none was present, the token MUST be absent in the Error message.

token

err-msg

supported-cipher-suites
The supported-cipher-suites parameter lists the ciphersuite(s) supported by the TEEP Agent. Details about the ciphersuite encoding can be found in Section 8. This otherwise optional parameter MUST be returned if err-code is ERR_UNSUPPORTED_CIPHER_SUITES.

supported-freshness-mechanisms
The supported-freshness-mechanisms parameter lists the freshness mechanism(s) supported by the TEEP Agent. Details about the encoding can be found in Section 9. This otherwise optional parameter MUST be returned if err-code is ERR_UNSUPPORTED_FRESHNESS_MECHANISMS.
versions
The versions parameter enumerates the TEEP protocol version(s) supported by the TEEP Agent. This otherwise optional parameter MUST be returned if err-code is ERR_UNSUPPORTED_MSG_VERSION.

suit-reports
If present, the suit-reports parameter contains a set of SUIT Reports as defined in Section 4 of [I-D.moran-suit-report]. If a token parameter was present in the Update message the Error message is in response to, the suit-report-nonce field MUST be present in the SUIT Report with a value matching the token parameter in the Update message.

err-code
The err-code parameter contains one of the error codes listed below). Only selected values are applicable to each message.

This specification defines the following initial error messages:

ERR_PERMANENT_ERROR (1)
The TEEP request contained incorrect fields or fields that are inconsistent with other fields. For diagnosis purposes it is RECOMMENDED to identify the failure reason in the error message. A TAM receiving this error might refuse to communicate further with the TEEP Agent for some period of time until it has reason to believe it is worth trying again, but it should take care not to give up on communication when there is no attestation evidence indicating that the error is genuine. In contrast, ERR_TEMPORARY_ERROR is an indication that a more aggressive retry is warranted.

ERR_UNSUPPORTED_EXTENSION (2)
The TEEP Agent does not support an extension included in the request message. For diagnosis purposes it is RECOMMENDED to identify the unsupported extension in the error message. A TAM receiving this error might retry the request without using extensions.

ERR_UNSUPPORTED_FRESHNESS_MECHANISMS (3)
The TEEP Agent does not support any freshness algorithm mechanisms in the request message. A TAM receiving this error might retry the request using a different set of supported freshness mechanisms in the request message.

ERR_UNSUPPORTED_MSG_VERSION (4)
The TEEP Agent does not support the TEEP protocol version indicated in the request message. A TAM receiving this error might retry the request using a different TEEP protocol version.
ERR_UNSUPPORTED_CIPHER_SUITES (5)
The TEEP Agent does not support any ciphersuites indicated in the request message. A TAM receiving this error might retry the request using a different set of supported ciphersuites in the request message.

ERR_BAD_CERTIFICATE (6)
Processing of a certificate failed. For diagnosis purposes it is RECOMMENDED to include information about the failing certificate in the error message. For example, the certificate was of an unsupported type, or the certificate was revoked by its signer. A TAM receiving this error might attempt to use an alternate certificate.

ERR_CERTIFICATE_EXPIRED (9)
A certificate has expired or is not currently valid. A TAM receiving this error might attempt to renew its certificate before using it again.

ERR_TEMPORARY_ERROR (10)
A miscellaneous temporary error, such as a memory allocation failure, occurred while processing the request message. A TAM receiving this error might retry the same request at a later point in time.

ERR_MANIFEST_PROCESSING_FAILED (17)
The TEEP Agent encountered one or more manifest processing failures. If the suit-reports parameter is present, it contains the failure details. A TAM receiving this error might still attempt to install or update other components that do not depend on the failed manifest.

New error codes should be added sparingly, not for every implementation error. That is the intent of the err-msg field, which can be used to provide details meaningful to humans. New error codes should only be added if the TAM is expected to do something behaviorally different upon receipt of the error message, rather than just logging the event. Hence, each error code is responsible for saying what the behavioral difference is expected to be.

5. EAT Profile

The TEEP protocol operates between a TEEP Agent and a TAM. While the TEEP protocol does not require use of EAT, use of EAT is encouraged and Section 4.3 explicitly defines a way to carry an Entity Attestation Token evidence in a QueryResponse.
As discussed in Section 4.3.1, the content of attestation evidence is opaque to the TEEP architecture, but the content of Attestation Results is not, where Attestation Results flow between a Verifier and a TAM (as the Relying Party). Although Attestation Results required by a TAM are separable from the TEEP protocol per se, this section is included as part of the requirements for building a compliant TAM that uses EATs for Attestation Results.

Section 7 of [I-D.ietf-rats-eat] defines the requirement for Entity Attestation Token profiles. This section defines an EAT profile for use with TEEP.

  (RFC-editor: upon RFC publication, replace string with "https://www.rfc-editor.org/info/rfcXXXX" where XXXX is the RFC number of this document.)

* Use of JSON, CBOR, or both: CBOR only.

* CBOR Map and Array Encoding: Only definite length arrays and maps.

* CBOR String Encoding: Only definite-length strings are allowed.

* CBOR Preferred Serialization: Encoders must use preferred serialization, and decoders need not accept non-preferred serialization.

* COSE/JOSE Protection: See Section 8.

* Detached EAT Bundle Support: DEB use is permitted.

* Verification Key Identification: COSE Key ID (kid) is used, where the key ID is the hash of a public key (where the public key may be used as a raw public key, or in a certificate).

* Endorsement Identification: Optional, but semantics are the same as in Verification Key Identification.

* Freshness: See Section 9.

* Required Claims: None.

* Prohibited Claims: None.
* Additional Claims: Optional claims are those listed in Section 4.3.1.

* Refined Claim Definition: None.

* CBOR Tags: CBOT Tags are not used.

* Manifests and Software Evidence Claims: The sw-name claim for a Trusted Component holds the URI of the SUIT manifest for that component.

6. Mapping of TEEP Message Parameters to CBOR Labels

In COSE, arrays and maps use strings, negative integers, and unsigned integers as their keys. Integers are used for compactness of encoding. Since the word "key" is mainly used in its other meaning, as a cryptographic key, this specification uses the term "label" for this usage as a map key.

This specification uses the following mapping:
<table>
<thead>
<tr>
<th>Name</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>supported-cipher-suites</td>
<td>1</td>
</tr>
<tr>
<td>challenge</td>
<td>2</td>
</tr>
<tr>
<td>version</td>
<td>3</td>
</tr>
<tr>
<td>selected-cipher-suite</td>
<td>5</td>
</tr>
<tr>
<td>selected-version</td>
<td>6</td>
</tr>
<tr>
<td>evidence</td>
<td>7</td>
</tr>
<tr>
<td>tc-list</td>
<td>8</td>
</tr>
<tr>
<td>ext-list</td>
<td>9</td>
</tr>
<tr>
<td>manifest-list</td>
<td>10</td>
</tr>
<tr>
<td>msg</td>
<td>11</td>
</tr>
<tr>
<td>err-msg</td>
<td>12</td>
</tr>
<tr>
<td>evidence-format</td>
<td>13</td>
</tr>
<tr>
<td>requested-tc-list</td>
<td>14</td>
</tr>
<tr>
<td>unneeded-tc-list</td>
<td>15</td>
</tr>
<tr>
<td>component-id</td>
<td>16</td>
</tr>
<tr>
<td>tc-manifest-sequence-number</td>
<td>17</td>
</tr>
<tr>
<td>have-binary</td>
<td>18</td>
</tr>
<tr>
<td>suit-reports</td>
<td>19</td>
</tr>
<tr>
<td>token</td>
<td>20</td>
</tr>
<tr>
<td>supported-freshness-mechanisms</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 2
7. Behavior Specification

Behavior is specified in terms of the conceptual APIs defined in section 6.2.1 of [I-D.ietf-teep-architecture].

7.1. TAM Behavior

When the ProcessConnect API is invoked, the TAM sends a QueryRequest message.

When the ProcessTeepMessage API is invoked, the TAM first does validation as specified in Section 4.1.2, and drops the message if it is not valid. Otherwise, it proceeds as follows.

If the message includes a token, it can be used to match the response to a request previously sent by the TAM. The TAM MUST expire the token value after receiving the first response from the device that has a valid signature and ignore any subsequent messages that have the same token value. The token value MUST NOT be used for other purposes, such as a TAM to identify the devices and/or a device to identify TAMs or Trusted Components.

If a QueryResponse message is received that contains evidence, the evidence is passed to an attestation Verifier (see [I-D.ietf-rats-architecture]) to determine whether the Agent is in a trustworthy state. Based on the results of attestation, and the lists of installed, requested, and unneeded Trusted Components reported in the QueryResponse, the TAM determines, in any implementation specific manner, which Trusted Components need to be installed, updated, or deleted, if any. If any Trusted Components need to be installed, updated, or deleted, the TAM sends an Update message containing SUIT Manifests with command sequences to do the relevant installs, updates, or deletes. It is important to note that the TEEP Agent’s Update Procedure requires resolving and installing any dependencies indicated in the manifest, which may take some time, and the resulting Success or Error message is generated only after completing the Update Procedure. Hence, depending on the freshness mechanism in use, the TAM may need to store data (e.g., a nonce) for some time.

If a Success or Error message is received containing one or more SUIT Reports, the TAM also validates that the nonce in any SUIT Report matches the token sent in the Update message, and drops the message if it does not match. Otherwise, the TAM handles the update in any implementation specific way, such as updating any locally cached information about the state of the TEEP Agent, or logging the results.
If any other Error message is received, the TAM can handle it in any implementation specific way, but Section 4.6 provides recommendations for such handling.

7.2. TEEP Agent Behavior

When the RequestTA API is invoked, the TEEP Agent first checks whether the requested TA is already installed. If it is already installed, the TEEP Agent passes no data back to the caller. Otherwise, if the TEEP Agent chooses to initiate the process of requesting the indicated TA, it determines (in any implementation specific way) the TAM URI based on any TAM URI provided by the RequestTA caller and any local configuration, and passes back the TAM URI to connect to. It MAY also pass back a QueryResponse message if all of the following conditions are true:

* The last QueryRequest message received from that TAM contained no token or challenge,

* The ProcessError API was not invoked for that TAM since the last QueryResponse message was received from it, and

* The public key or certificate of the TAM is cached and not expired.

When the RequestPolicyCheck API is invoked, the TEEP Agent decides whether to initiate communication with any trusted TAMs (e.g., it might choose to do so for a given TAM unless it detects that it has already communicated with that TAM recently). If so, it passes back a TAM URI to connect to. If the TEEP Agent has multiple TAMs it needs to connect with, it just passes back one, with the expectation that RequestPolicyCheck API will be invoked to retrieve each one successively until there are no more and it can pass back no data at that time. Thus, once a TAM URI is returned, the TEEP Agent can remember that it has already initiated communication with that TAM.

When the ProcessError API is invoked, the TEEP Agent can handle it in any implementation specific way, such as logging the error or using the information in future choices of TAM URI.

When the ProcessTeepMessage API is invoked, the Agent first does validation as specified in Section 4.1.2, and drops the message if it is not valid. Otherwise, processing continues as follows based on the type of message.

When a QueryRequest message is received, the Agent responds with a QueryResponse message if all fields were understood, or an Error message if any error was encountered.
When an Update message is received, the Agent attempts to update the Trusted Components specified in the SUIT manifests by following the Update Procedure specified in [I-D.ietf-suit-manifest], and responds with a Success message if all SUIT manifests were successfully installed, or an Error message if any error was encountered. It is important to note that the Update Procedure requires resolving and installing any dependencies indicated in the manifest, which may take some time, and the Success or Error message is generated only after completing the Update Procedure.

8. Ciphersuites

The TEEP protocol uses COSE for protection of TEEP messages. After a QueryResponse is received, the selected cryptographic algorithm is used in subsequent TEEP messages (Install, Success, and Error). To negotiate cryptographic mechanisms and algorithms, the TEEP protocol defines the following ciphersuite structure.

ciphersuite = [  
  teep-cose-sign-algs / nil,  
  teep-cose-encrypt-algs / nil ,  
  teep-cose-mac-algs / nil  
]

The ciphersuite structure is used to present the combination of mechanisms and cryptographic algorithms. Each suite value corresponds with a COSE-type defined in Section 2 of [RFC8152].

supported-cipher-suites = [ + suite ]

Cryptographic algorithm values are defined in the COSE Algorithms registry [COSE.Algorithm]. A TAM MUST support both of the following ciphersuites. A TEEP Agent MUST support at least one of the two but can choose which one. For example, a TEEP Agent might choose a given ciphersuite if it has hardware support for it.

tee-cose-sign-algs /= cose-alg-es256,  
tee-cose-sign-algs /= cose-alg-eddsa

A TAM or TEEP Agent MUST also support the following algorithms:

tee-cose-encrypt-algs /= cose-alg-accm-16-64-128  
tee-cose-mac-algs /= cose-alg-hmac-256

A TAM or TEEP Agent MAY also support one or more of the following algorithms:
teep-cose-sign-algs /= cose-alg-ps256,
teep-cose-sign-algs /= cose-alg-ps384,
teep-cose-sign-algs /= cose-alg-ps512,
teep-cose-sign-algs /= cose-alg-rsa-oaep-256,
teep-cose-sign-algs /= cose-alg-rsa-oaep-512

Any ciphersuites without confidentiality protection can only be added if the associated specification includes a discussion of security considerations and applicability, since manifests may carry sensitive information. For example, Section 6 of [I-D.ietf-teep-architecture] permits implementations that terminate transport security inside the TEE and if the transport security provides confidentiality then additional encryption might not be needed in the manifest for some use cases. For most use cases, however, manifest confidentiality will be needed to protect sensitive fields from the TAM as discussed in Section 9.8 of [I-D.ietf-teep-architecture].

9. Freshness Mechanisms

A freshness mechanism determines how a TAM can tell whether evidence provided in a Query Response is fresh. There are multiple ways this can be done as discussed in Section 10 of [I-D.ietf-rats-architecture].

Each freshness mechanism is identified with an integer value, which corresponds to an IANA registered freshness mechanism (see Section 11.2. This document defines the following freshness mechanisms:

<table>
<thead>
<tr>
<th>Value</th>
<th>Freshness mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nonce</td>
</tr>
<tr>
<td>2</td>
<td>Timestamp</td>
</tr>
<tr>
<td>3</td>
<td>Epoch ID</td>
</tr>
</tbody>
</table>

Table 3

In the Nonce mechanism, the evidence MUST include a nonce provided in the QueryRequest challenge. In other mechanisms, a timestamp or epoch ID determined via mechanisms outside the TEEP protocol is used, and the challenge is only needed in the QueryRequest message if a challenge is needed in generating evidence for reasons other than freshness.
If a TAM supports multiple freshness mechanisms that require different challenge formats, the QueryRequest message can currently only send one such challenge. This situation is expected to be rare, but should it occur, the TAM can choose to prioritize one of them and exclude the other from the supported-freshness-mechanisms in the QueryRequest, and resend the QueryRequest with the other mechanism if an ERR_UNSUPPORTED_FRESHNESS_MECHANISMS Error is received that indicates the TEEP Agent supports the other mechanism.

10. Security Considerations

This section summarizes the security considerations discussed in this specification:

Cryptographic Algorithms

TEEP protocol messages exchanged between the TAM and the TEEP Agent are protected using COSE. This specification relies on the cryptographic algorithms provided by COSE. Public key based authentication is used by the TEEP Agent to authenticate the TAM and vice versa.

Attestation

A TAM can rely on the attestation evidence provided by the TEEP Agent. To sign the attestation evidence, it is necessary for the device to possess a public key (usually in the form of a certificate [RFC5280]) along with the corresponding private key. Depending on the properties of the attestation mechanism, it is possible to uniquely identify a device based on information in the attestation evidence or in the certificate used to sign the attestation evidence. This uniqueness may raise privacy concerns. To lower the privacy implications the TEEP Agent MUST present its attestation evidence only to an authenticated and authorized TAM and when using EATS, it SHOULD use encryption as discussed in [I-D.ietf-rats-eat], since confidentiality is not provided by the TEEP protocol itself and the transport protocol under the TEEP protocol might be implemented outside of any TEE. If any mechanism other than EATs is used, it is up to that mechanism to specify how privacy is provided.
Trusted Component Binaries
Each Trusted Component binary is signed by a Trusted Component Signer. It is the responsibility of the TAM to relay only verified Trusted Components from authorized Trusted Component Signers. Delivery of a Trusted Component to the TEEP Agent is then the responsibility of the TAM, using the security mechanisms provided by the TEEP protocol. To protect the Trusted Component binary, the SUIT manifest format is used and it offers a variety of security features, including digital signatures and symmetric encryption.

Personalization Data
A Trusted Component Signer or TAM can supply personalization data along with a Trusted Component. This data is also protected by a SUIT manifest. Personalization data signed and encrypted by a Trusted Component Signer other than the TAM is opaque to the TAM.

TEEP Broker
As discussed in section 6 of [I-D.ietf-teep-architecture], the TEEP protocol typically relies on a TEEP Broker to relay messages between the TAM and the TEEP Agent. When the TEEP Broker is compromised it can drop messages, delay the delivery of messages, and replay messages but it cannot modify those messages. (A replay would be, however, detected by the TEEP Agent.) A compromised TEEP Broker could reorder messages in an attempt to install an old version of a Trusted Component. Information in the manifest ensures that TEEP Agents are protected against such downgrade attacks based on features offered by the manifest itself.

Trusted Component Signer Compromise
A TAM is responsible for vetting a Trusted Component and before distributing them to TEEP Agents.

It is RECOMMENDED to provide a way to update the trust anchor store used by the TEE, for example using a firmware update mechanism. Thus, if a Trusted Component Signer is later compromised, the TAM can update the trust anchor store used by the TEE, for example using a firmware update mechanism.

CA Compromise
The CA issuing certificates to a TEE or a Trusted Component Signer might get compromised. It is RECOMMENDED to provide a way to update the trust anchor store used by the TEE, for example using a firmware update mechanism. If the CA issuing certificates to devices gets compromised then these devices might be rejected by a TAM, if revocation is available to the TAM.
TAM Certificate Expiry
The integrity and the accuracy of the clock within the TEE determines the ability to determine an expired TAM certificate, if certificates are used.

Compromised Time Source
As discussed above, certificate validity checks rely on comparing validity dates to the current time, which relies on having a trusted source of time, such as [RFC8915]. A compromised time source could thus be used to subvert such validity checks.

11. IANA Considerations

11.1. Media Type Registration

IANA is requested to assign a media type for application/teep+cbor.

Type name: application
Subtype name: teep+cbor
Required parameters: none
Optional parameters: none
Encoding considerations: Same as encoding considerations of application/cbor.
Security considerations: See Security Considerations Section of this document.
Interoperability considerations: Same as interoperability considerations of application/cbor as specified in [RFC7049].
Published specification: This document.
Applications that use this media type: TEEP protocol implementations
Fragment identifier considerations: N/A
Additional information: Deprecated alias names for this type: N/A
    Magic number(s): N/A
    File extension(s): N/A
    Macintosh file type code(s): N/A
11.2. Freshness Mechanism Registry

IANA is also requested to create a new registry for freshness mechanisms.

Name of registry: TEEP Freshness Mechanisms

Policy: Specification Required [RFC8126]

Additional requirements: The specification must document relevant security considerations.

Initial values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Freshness mechanism</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nonce</td>
<td>RFC TBD Section 9</td>
</tr>
<tr>
<td>2</td>
<td>Timestamp</td>
<td>RFC TBD Section 9</td>
</tr>
<tr>
<td>3</td>
<td>Epoch ID</td>
<td>RFC TBD Section 9</td>
</tr>
</tbody>
</table>

Table 4

(RFC Editor: please replace TBD above with the number assigned to this document.)

12. References

12.1. Normative References

[COSE.Algorithm]
[I-D.ietf-rats-architecture]

[I-D.ietf-rats-eat]

[I-D.ietf-suit-manifest]

[I-D.moran-suit-report]

[I-D.moran-suit-trust-domains]


[RFC5280]  Cooper, D., Santesson, S., Farrell, S., Boeyen, S.,
         Housley, R., and W. Polk, "Internet X.509 Public Key
         Infrastructure Certificate and Certificate Revocation List

[RFC7049]  Bormann, C. and P. Hoffman, "Concise Binary Object
         Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049,

[RFC8152]  Schaad, J., "CBOR Object Signing and Encryption (COSE)",
         RFC 8152, DOI 10.17487/RFC8152, July 2017,
         <https://www.rfc-editor.org/info/rfc8152>.

[RFC8174]  Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC
         2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174,

12.2.  Informative References

[I-D.birkholz-rats-suit-claims]
Birkholz, H. and B. Moran, "Trustworthiness Vectors for
the Software Updates of Internet of Things (SUIT) Workflow
Model", Work in Progress, Internet-Draft, draft-birkholz-
rats-suit-claims-03, 12 January 2022,
<https://www.ietf.org/archive/id/draft-birkholz-rats-suit-
claims-03.txt>.

[I-D.ietf-teep-architecture]
Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler,
"Trusted Execution Environment Provisioning (TEEP)
Architecture", Work in Progress, Internet-Draft, draft-ietf-teep-
archnitecture-16, 28 February 2022,
<https://www.ietf.org/archive/id/draft-ietf-teep-
archnitecture-16.txt>.

Writing an IANA Considerations Section in RFCs", BCP 26,
RFC 8126, DOI 10.17487/RFC8126, June 2017,

[RFC8610]  Birkholz, H., Vigano, C., and C. Bormann, "Concise Data
Definition Language (CDDL): A Notational Convention to
Express Concise Binary Object Representation (CBOR) and
JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610,
A. Contributors

We would like to thank Brian Witten (Symantec), Tyler Kim (Solacia), Nick Cook (Arm), and Minho Yoo (IoTrust) for their contributions to the Open Trust Protocol (OTrP), which influenced the design of this specification.

B. Acknowledgements

We would like to thank Eve Schooler for the suggestion of the protocol name.

We would like to thank Kohei Isobe (TRASIO/SECOM), Ken Takayama (SECOM) Kuniyasu Suzuki (TRASIO/AIST), Tsukasa Oi (TRASIO), and Yuichi Takita (SECOM) for their valuable implementation feedback.

We would also like to thank Carsten Bormann and Henk Birkholz for their help with the CDDL.

C. Complete CDDL

Valid TEEP messages MUST adhere to the following CDDL data definitions, except that SUIT_Envelope and SUIT_Component_Identifier are specified in [I-D.ietf-suit-manifest].

```
teep-message = $teep-message-type .within teep-message-framework

SUIT_Envelope = any

teep-message-framework = [  
  type: uint (0..23) / $teep-type-extension,  
  options: { * teep-option },  
  * uint; further integers, e.g., for data-item-requested  
]

teep-option = (uint => any)

; messages defined below:
$teep-message-type /= query-request  
$teep-message-type /= query-response  
$teep-message-type /= update  
$teep-message-type /= teep-success  
$teep-message-type /= teep-error
```
; message type numbers, uint (0..23)
TEEP-TYPE-query-request = 1
TEEP-TYPE-query-response = 2
TEEP-TYPE-update = 3
TEEP-TYPE-teep-success = 5
TEEP-TYPE-teep-error = 6

version = .within uint .size 4
ext-info = .within uint .size 4

; data items as bitmaps
data-item-requested = $data-item-requested .within uint .size 8
attestation = 1
$data-item-requested /= attestation
trusted-components = 2
$data-item-requested /= trusted-components
extensions = 4
$data-item-requested /= extensions

query-request = [ 
  type: TEEP-TYPE-query-request, 
  options: { 
    ? token => bstr .size (8..64),
    ? supported-cipher-suites => [ + suite ],
    ? supported-freshness-mechanisms => [ + freshness-mechanism ],
    ? challenge => bstr .size (8..512),
    ? versions => [ + version ],
    * $$query-request-extensions
    * $$teep-option-extensions
  },
  data-item-requested: data-item-requested
]

; ciphersuites
suite = [ 
  teep-cose-sign-algs / nil,
  teep-cose-encrypt-algs / nil,
  teep-cose-mac-algs / nil
]

teep-cose-sign-algs /= cose-alg-es256,
teep-cose-sign-algs /= cose-alg-eddsa
teepe-cose-sign-algs /= cose-alg-ps256,
teep-cose-sign-algs /= cose-alg-ps384,
teep-cose-sign-algs /= cose-alg-ps512,
teep-cose-sign-algs /= cose-alg-rsa-oaep-256,
teep-cose-sign-algs /= cose-alg-rsa-oaep-512
teep-cose-encrypt-algs /= cose-alg-accm-16-64-128

teep-cose-mac-algs /= cose-alg-hmac-256

; algorithm identifiers defined in the IANA COSE Algorithms Registry
cose-alg-es256 = -7
cose-alg-eddsa = -8
cose-alg-ps256 = -37
cose-alg-ps384 = -38
cose-alg-ps512 = -39
cose-alg-rsa-oaep-256 = -41
cose-alg-rsa-oaep-512 = -42
cose-alg-accm-16-64-128 = 10
cose-alg-hmac-256 = 5

; freshness-mechanisms

freshness-mechanism = $TEEP-freshness-mechanism .within uint .size 4

FRESHNESS_NONCE = 0
FRESHNESS_TIMESTAMP = 1
FRESHNESS_EPOCH_ID = 2

$TEEP-freshness-mechanism /= FRESHNESS_NONCE
$TEEP-freshness-mechanism /= FRESHNESS_TIMESTAMP
$TEEP-freshness-mechanism /= FRESHNESS_EPOCH_ID

query-response = [  
  type: TEEP-TYPE-query-response,
  options: {  
    ? token => bstr .size (8..64),
    ? selected-cipher-suite => suite,
    ? selected-version => version,
    ? evidence-format => text,
    ? evidence => bstr,
    ? tc-list => [ + tc-info ],
    ? requested-tc-list => [ + requested-tc-info ],
    ? unneeded-tc-list => [ + SUIT_Component_Identifier ],
    ? ext-list => [ + ext-info ],
    * $$query-response-extensions,
    * $$teep-option-extensions
  }
]

tc-info = {  
  component-id => SUIT_Component_Identifier,
  ? tc-manifest-sequence-number => .within uint .size 8
}
request-tc-info = {
  component-id => SUIT_Component_Identifier,
  ? tc-manifest-sequence-number => .within uint .size 8
  ? have-binary => bool
}

update = [
  type: TEEP-TYPE-update,
  options: {
    ? token => bstr .size (8..64),
    ? manifest-list => [ + bstr .cbor SUIT_Envelope ],
  * $$update-extensions,
  * $$teep-option-extensions
  }
]

teep-success = [
  type: TEEP-TYPE-teep-success,
  options: {
    ? token => bstr .size (8..64),
    ? msg => text .size (1..128),
    ? suit-reports => [ + suit-report ],
  * $$teep-success-extensions,
  * $$teep-option-extensions
  }
]

teep-error = [
  type: TEEP-TYPE-teep-error,
  options: {
    ? token => bstr .size (8..64),
    ? err-msg => text .size (1..128),
    ? supported-cipher-suites => [ + suite ],
    ? supported-freshness-mechanisms => [ + freshness-mechanism ],
    ? versions => [ + version ],
    ? suit-reports => [ + suit-report ],
  * $$teep-error-extensions,
  * $$teep-option-extensions
 },
  err-code: uint (0..23)
]

; The err-code parameter, uint (0..23)
ERR_PERMANENT_ERROR = 1
ERR_UNSUPPORTED_EXTENSION = 2
ERR_UNSUPPORTED_FRESHNESS_MECHANISMS = 3
ERR_UNSUPPORTED_MSG_VERSION = 4
ERR_UNSUPPORTED_CIPHER_SUITES = 5
ERR_BAD_CERTIFICATE = 6
ERR_CERTIFICATE_EXPIRED = 9
ERR_TEMPORARY_ERROR = 10
ERR_MANIFEST_PROCESSING_FAILED = 17

; labels of mapkey for teep message parameters, uint (0..23)
supported-cipher-suites = 1
challenge = 2
versions = 3
selected-cipher-suite = 5
selected-version = 6
evidence = 7
tc-list = 8
ext-list = 9
manifest-list = 10
msg = 11
err-msg = 12
evidence-format = 13
requested-tc-list = 14
unneeded-tc-list = 15
component-id = 16
tc-manifest-sequence-number = 17
have-binary = 18
suit-reports = 19
token = 20
supported-freshness-mechanisms = 21

D.  Examples of Diagnostic Notation and Binary Representation

This section includes some examples with the following assumptions:

* The device will have two TCs with the following SUIT Component Identifiers:
  - [ 0x000102030405060708090a0b0c0d0e0f ]
  - [ 0x100102030405060708090a0b0c0d0e0f ]

* SUIT manifest-list is set empty only for example purposes (see Appendix E for actual manifest examples)

D.1.  QueryRequest Message

D.1.1.  CBOR Diagnostic Notation
/ query-request = /
[
  1, / type : TEEP-TYPE-query-request = 1 (uint (0..23)) /
  / options : /
  {
    20 : 0xa0a1a2a3a4a5a6a7a8a9aaabacadaeaf,
    / token = 20 (mapkey) :
      h’a0a1a2a3a4a5a6a7a8a9aaabacadaeaf’ (bstr .size (8..64)),
      generated by TAM /
    1 : [ 1 ], / supported-cipher-suites = 1 (mapkey) :
      TEEP-AES-CCM-16-64-128-HMAC256--256-X25519-EdDSA =
      [ 1 ] (array of .within uint .size 4) /
    3 : [ 0 ] / version = 3 (mapkey) :
      [ 0 ] (array of .within uint .size 4) /
  },
  3 / data-item-requested :
    attestation | trusted-components = 3 (.within uint .size 8) /
]
/ eat-claim-set = /
{
  / issuer / 1: "joe",
  / timestamp (iat) / 6: 1(1526542894)
  / nonce / 10: h'948f8860d13a463e8e',
  / secure-boot / 15: true,
  / debug-status / 16: 3, / disabled-permanently /
  / security-level / 14: 3, / secure-restricted /
  / device-identifier / <TBD>: h'e99600dd921649798b013e9752dcf0c5',
  / vendor-identifier / <TBD>: h'2b03879b33434a7ca682b8af84c19fd4',
  / class-identifier / <TBD>: h'9714a5796bd245a3a4ab4f977cb8487f',
  / chip-version / 26: [ "MyTEE", 1 ],
  / component-identifier / <TBD>: h'60822887d35e43d5b603d18bcaa3f08d',
  / version / <TBD>: "v0.1"
}

D.3. QueryResponse Message

D.3.1. CBOR Diagnostic Notation
/ query-response = /
[
  2, / type : TEEP-TYPE-query-response = 2 (uint (0..23)) /
  / options : /
  { 
    20 : 0xa0a2a3a4a5a6a7a8a9aaabacadaeaf, 
    / token = 20 (mapkey) :
        h’0xa0a2a3a4a5a6a7a8a9aaabacadaeaf’ (bstr .size (8..64)),
        given from TAM’s QueryRequest message /
    5 : 1, / selected-cipher-suite = 5 (mapkey) :
        TEEP-AES-CCM-16-64-128-HMAC256--256-X25519-EdDSA =
        1 (.within uint .size 4) /
    6 : 0, / selected-version = 6 (mapkey) :
        0 (.within uint .size 4) /
    7 : ... / evidence = 7 (mapkey) :
        Entity Attestation Token /
    8 : [ / tc-list = 8 (mapkey) : (array of tc-info) /
      { 
        16 : [ 0x000102030405060708090a0b0c0d0e0f ] / component-id =
        16 (mapkey) : [ h’000102030405060708090a0b0c0d0e0f’ ]
        (SUIT_Component_Identifier = [* bstr]) /
      },
      { 
        16 : [ 0x100102030405060708090a0b0c0d0e0f ] / component-id =
        16 (mapkey) : [ h’100102030405060708090a0b0c0d0e0f’ ]
        (SUIT_Component_Identifier = [* bstr]) /
      }
    ]
  }
]

D.3.2. CBOR Binary Representation
D.4. Update Message

D.4.1. CBOR Diagnostic Notation

/update = /
[3, / type : TEEP-TYPE-update = 3 (uint (0..23)) /
/ options : /
{20 : 0xa0a1a2a3a4a5a6a7a8a9aaabacadaeaf,
/ token = 20 (mapkey) :
'\x00a0a1a2a3a4a5a6a7a8a9aaabacadaeaf' (bstr .size (8..64)),
generated by TAM /
10 : [ ] / manifest-list = 10 (mapkey) :
[ ] (array of bstr wrapped SUIT_Envelope(any)) /
/ empty, example purpose only /
}
]

D.4.2. CBOR Binary Representation
D.5. Success Message

D.5.1. CBOR Diagnostic Notation

```plaintext
/ teep-success = /
[
  5,  / type : TEEP-TYPE-teep-success = 5 (uint (0..23)) /
  / options : /
  {
    20 : 0xa0a1a2a3a4a5a6a7a8a9aaabacadaeaf,
    / token = 20 (mapkey) :
    h’a0a1a2a3a4a5a6a7a8a9aaabacadaeaf’ (bstr .size (8..64)),
    given from TAM’s Update message /
  }
]
```

D.5.2. CBOR Binary Representation

```plaintext
82  # array(2)
  05  # unsigned(5) uint (0..23)
  A1  # map(1)
  14  # unsigned(20) uint (0..23)
  4F  # bytes(16) (8..64)
     A0A1A2A3A4A5A6A7A8A9AAABACADAEBF
```

D.6. Error Message

D.6.1. CBOR Diagnostic Notation

```plaintext
```

/ teep-error = /
[
  6, / type : TEEP-TYPE-teep-error = 6 (uint (0..23)) /
  / options : /
    { 20 : 0xa0a1a2a3a4a5a6a7a8a9aaabacadaef, / token = 20 (mapkey) :
     h’a0a1a2a3a4a5a6a7a8a9aaabacadaef’ (bstr .size (8..64)),
     given from TAM’s Update message /
    12 : "disk-full" / err-msg = 12 (mapkey) :
     "disk-full" (text .size (1..128)) /
  },
  17, / err-code : ERR_MANIFEST_PROCESSING_FAILED = 17 (uint (0..23)) /
]

D.6.2. CBOR binary Representation

83 # array(3)
  06 # unsigned(6) uint (0..23)
  A2 # map(2)
    14 # unsigned(20) uint (0..23)
    4F # bytes(16) (8..64)
     A0A1A2A3A4A5A6A7A8A9AAABACADAEEAF
  0C # unsigned(12) uint (0..23)
  69 # text(9) (1..128)
    6469736B2D66756C64 # "disk-full"
  11 # unsigned(17) uint (0..23)

E. Examples of SUIT Manifests

This section shows some examples of SUIT manifests described in Section 4.4.

The examples are signed using the following ECDSA secp256r1 key with SHA256 as the digest function.

COSE_Sign1 Cryptographic Key:

-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwIBAQgApZYjZCUGLM50VBC
CjYStx+09jGmnyJPrpDLTz/hiXOhRANCAASEloEarguqq9JhVxie7NomvqqL8Rtv
P+bitWChdvArTsftksCJExwKtkrNHX19OB3N+wnAUtszmR23M4tK1W
-----END PRIVATE KEY-----

The corresponding public key can be used to verify these examples:
Example 1: SUIT Manifest pointing to URI of the Trusted Component Binary

CBOR Diagnostic Notation of SUIT Manifest

```
/ SUIT_Envelope_Tagged / 107( {
  / suit-authentication-wrapper / 2: << [<<
    / suit-digest-algorithm-id: / -16 / suit-cose-alg-sha256 /,
    / suit-digest-bytes: / h'DB601ADE73092B58532CA03FBB663DE49532435336F1558B49BB622726A2FEDD'
  ] >>,
  / suit-manifest / 3: << {
    / suit-manifest-version / 1: 1,
    / suit-manifest-sequence-number / 2: 3,
    / suit-common / 3: << {
      / suit-components / 2: [
        h'544545502D446576696365',           / "TEEP-Device" /
        h'5365637572654665',                 / "SecureFS" /
        h'8D82573A926D4754935332DC29997F74', / tc-uuid /
        h'7461'                              / "ta" /
      ]
    }
  }
}) >>
```

Tschofenig, et al. Expires 8 September 2022
Internet-Draft                TEEP Protocol                   March 2022

} >>
} >>,

/suit-install / 9: << [ 
/suit-directive-override-parameters / 20, { 
/suit-parameter-uri / 21: "https://example.org/8d82573a-926d-4754-9353-32dc9997f74.ta"
},
/suit-directive-fetch / 21, 15,
/suit-condition-image-match / 3, 15
] >>
} >>
)
)

CBOR Binary Representation

D8 6B                                               # tag(107) / SUIT_Envelope_Ta
gged /

A2                                               # map(2)

cation-wrapper /

02                                            # unsigned(2) / suit-authenti
cation-wrapper /

58 73                                         # bytes(115)

82                                         # array(2)

58 24                                   # bytes(36)

82                                   # array(2)

2F                                # negative(15) / -16 = suit-c
ose-alg-sha256 /

58 20                                          # bytes(32)

DB601ADE73092B58532CA03FBB663DE49532435336F1558B49BB622726A2

FEDD                                          # bytes(74)

58 4A                                         # tag(108) / COSE_Sign1_Tagged

D2                                          # map(0)

84                                         # array(4)

43                                         # bytes(3)

A1                                             # map(1)

01                                         # unsigned(1) / algorithm-id

/ 26                                         # negative(6) / -7 = ES256 /

A0                                             # map(0)

F6                                             # primitive(22) / null /

58 40                                        # bytes(64)

5B2D535A2B6D53C585C1074F414DA9E10BD285C99A33916D5D3E25388
812504817AC48B62B8E984EC622785BD1C411888BE531B159450781B201F6F28579A4

03                                        # unsigned(3) / suit-manifest:

/ 58 D4                                      # bytes(212)

A4                                          # map(4)

01                                          # unsigned(1) / suit-manifest

-version: / 01                                      # unsigned(1)

02                                      # unsigned(2) / suit-manifest

-sequence-number: / 03                                      # unsigned(3)

03                                      # unsigned(3) / suit-common:

/ 58 84                                      # bytes(132)

A2                                          # map(2)

02                                          # unsigned(2) / suit-componen
4B # bytes(11) "TEEP-Device"
48 # bytes(8) "SecureFS"
50 # bytes(16) tc-uuid
42 # bytes(2) "ta"
04 # unsigned(4) / suit-common-sequence: /
58 54 # bytes(84) array(6)
86 # unsigned(20) / suit-directive-override-parameters: /
14 # map(4)
A4 # unsigned(1) / suit-parameter-vendor-identifier: /
50 # bytes(16) C0DD5F15243566087DB4F5B0AA26C2F
02 # unsigned(2) / suit-parameter-class-identifier: /
50 # bytes(16) DB42F7093D8C55BAA8C5265FC5820F4E
03 # unsigned(3) / suit-parameter-image-digest: /
58 24 # bytes(36) array(2)
82 # negative(15) / -16 = suit-cose-alg-sha256 /
58 20 # bytes(32)
046397481469468ECE8 0E # unsigned(14) / suit-parameter-image-size: /
14 # unsigned(20) / suit-condition-vendor-identifier: /
01 # unsigned(1) / suit-condition-class-identifier: /
0F # unsigned(15) 
02 # unsigned(2) / suit-condition-image-digest: /
09 # unsigned(9) / suit-install:
/ 58 45 # bytes(69)
86 # array(6)
14 # unsigned(20) / suit-directive-override-parameters: /
A1 # map(1)
15 # unsigned(1) / suit-parameter-uri: /
78 3B # text(59)
68747470733A2F2F6578616D706C652E6F72672F3864383235733362D393236642D343735342D393335332D333264633239393376637342E7461 # "https://example.org/8d82573a-926d-4754-9353-32dc29997f74.ta"
15 # unsigned(21) / suit-directive-fetch: /
0F # unsigned(15)
03 # unsigned(3) / suit-condition
n-image-match: /

OF # unsigned(15)

CBOR Binary in Hex
Example 2: SUIT Manifest including the Trusted Component Binary

CBOR Diagnostic Notation of SUIT Manifest

```
/ SUIT_Envelope_Tagged / 107( {
  / suit-authentication-wrapper / 2: << [
    << [
      / suit-digest-algorithm-id: / -16 / suit-cose-alg-sha256 /,
      / suit-digest-bytes: / h'14A98BE957DE38FAE37376EA491FD6CAD9BFBD3C90051C8F5B017D7A496C3B05'
    ] >>,
    << / COSE_Sign1_Tagged / 18( [
      / protected: / << {
        / algorithm-id / 1: -7 / ES256 /,
      } >>,
      / unprotected: / {},
      / payload: / null,
      / signature: / h'4093B323953785981EB607C8BA61B21E5C4F85726A2AF48C1CB05BD4401B1B1565070728FDA38E6496D631E1D23F966CFF7805EDE721D48507D9192993DA8722'
    ] ) >>
  ] >>,
  / suit-integrated-payload / "#tc": h'48656C6C6F2C205365635726520576F726C6421',
  / "Hello, Secure World!" /
  / suit-manifest / 3: << {
    / suit-manifest-version / 1: 1,
    / suit-manifest-sequence-number / 2: 3,
    / suit-common / 3: << {
      / suit-components / 2: [ {
        h'544545502D446576696365', / "TEEP-Device" /,
        h'5365637572654653', / "SecureFS" /,
        h'8DB2573A926D4754935332DC29997F74', / tc-uuid /,
        h'7461' / "ta" /
      },
      / suit-common-sequence / 4: << [
        / suit-directive-override-parameters / 20, { / suit-parameter-vendor-identifier / 1: h'C0DDD5F15243566087DB4F5B0AA26C2F' /,
      }
    ]
  ]
}
```
Internet-Draft TEEP Protocol March 2022

/ suit-parameter-class-identifier / 2: h'DB42F7093D8C55BAA8C5265FC5820F
4E',
/ suit-parameter-image-digest / 3: << [
/ suit-digest-algorithm-id: / -16 / suit-cose-alg-sha256 /,
/ suit-digest-bytes: / h'8CF71AC86AF31BE184EC7A05A411A8C3A14FD9B77A30
D046397481469468ECE8'
] >>,
/ suit-parameter-image-size / 14: 20
},
/ suit-condition-vendor-identifier / 1, 15,
/ suit-condition-class-identifier / 2, 15
] >>
} >>,
/ suit-install / 9: << [
/ suit-directive-override-parameters / 20, {
/ suit-parameter-uri / 21: "#tc"
},
/ suit-directive-fetch / 21, 15,
/ suit-condition-image-match / 3, 15
] >>
} >>
)

CBOR Binary Representation

D8 6B                                               # tag(107) / SUIT_Envelope_Tagged
A3                                               # map(3)
  02                                            # unsigned(2) / suit-authentication-wrapper /
  58 73                                         # bytes(115)
  82                                         # array(2)
  58 24                                   # bytes(36)
  82                                   # array(2)
  2F                                # negative(15) / -16 = suit-cose-alg-sha256 /
  58 20                             # bytes(32)
       14A98BE957DE38FAE37376EA491FD6CAD9BFBD3C90051C8F5B017D7A496C
3B05
  58 4A                          # bytes(74)
  D2                            # tag(18) / COSE_Sign1_Tagged
  84                                         # array(4)
     43                             # bytes(3)
        A1                          # map(1)
           01                        # unsigned(1) / algorithm-id
/ 26                                # negative(6) / -7 = ES256 /
     A0                            # map(0)
        F6                            # primitive(22) / null /
           58 40                        # bytes(64)
           4093B323953785981EB607C8BA61B21E5C4F85726A2AF48C1CB05BD44
01B1B1565070728FADA38E6496D631E1D23F966CFF7805EDE721D48507D9192993DA8722
63                                         # text(3) / suit-integrated-payload /
  237463                              # "#tc"
  54                                         # bytes(20)
     48656C6C6F2C2053656375726520576F726C6421 "Hello, Secure World!"
03
: /
  58 9A
  A4
  01
  -version: /
    01
    02
  -sequence-number: /
    03
    03
  /
    58 84
    A2
    02
ts: /
  81
  84
  4B
  544545502D446576696365
  # "TEEP-Device"
  48
  5365637572654653
  # "SecureFS"
  50
  8D82573A926D4754935332DC29997F74
  # tc-uuid
  42
  7461
  04
sequence: /
  58 54
  86
  14
ve-override-parameters: /
  A4
  01
r-vendor-identifier: /
  50
  C0DDD5F15243566087DB4F5B0AA26C2F
  # bytes(16)
r-class-identifier: /
  50
  DB42F7093D8C55BAA8C5265FC5820F4E
  # bytes(16)
r-image-digest: /
  58 24
  82
  2F
  # negative(15) / -16 = suit-c
ose-alg-sha256 /
  58 20
  8CF71AC86AF31BE184EC7A05A411A8C3A14FD9B77A30D
046397481469468ECEB
er-image-size: /
  0E
  14
n-vendor-identifier: /
  0F
  02
n-class-identifier: /
Example 3: Supplying Personalization Data for Trusted Component Binary

CBOR Diagnostic Notation of SUIT Manifest

```
```
/ suit-dependency-digest / 1: [ 
  / suit-digest-algorithm-id: / -16 / suit-cose-alg-sha256 /, 
  / suit-digest-bytes: / h'F8690E5A86D010BF2B5348ABB99F2254DB7B608D0D626B98DB51AB3ECFC51907' 
] 
}, 
/ suit-components / 2: [ 
  h'544545502D446576696365', / "TEEP-Device" / 
  h'5365637572654653', / "SecureFS" / 
  h'636F6E6669672E6A736F6E' / "config.json" / 
], 
/ suit-common-sequence / 4: << 
  / suit-directive-set-component-index / 12, 0, 
  / suit-directive-override-parameters / 20, { 
    / suit-parameter-vendor-identifier / 1: h'C0DDD5F15243566087DB4F5B0AA26C2F', 
    / suit-parameter-class-identifier / 2: h'DB42F7093D8C55BAA8C5265FC5820F4E', 
    / suit-parameter-image-digest / 3: << [ 
      / suit-digest-algorithm-id: / -16 / suit-cose-alg-sha256 /, 
      / suit-digest-bytes: / h'AAABCCCDEEEFP00012223444566678889ABBCCDDEFFF01112333455567778999' 
    ] >>, 
    / suit-parameter-image-size / 14: 64 
  }, 
  / suit-condition-vendor-identifier / 1, 15, 
  / suit-condition-class-identifier / 2, 15 
} >>, 
/ suit-dependency-resolution / 7: << [ 
  / suit-directive-set-dependency-index / 13, 0, 
  / suit-directive-override-parameters / 20, { 
    / suit-parameter-uri / 21: "https://example.org/8d82573a-926d-4754-9353-32dc29997f74.suit" 
  }, 
  / suit-directive-fetch / 21, 2, 
  / suit-condition-image-match / 3, 15 
} >>, 
/ suit-install / 9: << [ 
  / suit-directive-set-dependency-index / 13, 0, 
  / suit-directive-process-dependency / 18, 0, 
  / suit-directive-set-component-index / 12, 0, 
  / suit-directive-override-parameters / 20, { 
    / suit-parameter-url / 21: "https://example.org/32dc29997f74.suit" 
  }, 
  / suit-directive-fetch / 21, 2, 
  / suit-condition-image-match / 3, 15 
} >>, 
/ suit-validate / 10: << [ 
  / suit-directive-set-component-index / 12, 0,
/ suit-condition-image-match/ 3, 15
] >>
} >>
}
)

CBOR Binary Represenation

D8 6B                                               # tag(107) / SUIT_Envelope_Ta
gado /
A2
02
cation-wrapper: /
58 73
82
58 24
82
2F
ose-alg-sha256 /
58 20
CE596D785169B72712560B3A246AA98F814498EA3625EEBB72CED9AF273E
7FD
58 4A
D2
/
84
43
A1
01
/
26
A0
F6
58 40
E9083A71D2BFC352853037B9C3116A5EDF23BE0F4B4357A8A835F724
660DA7482C64345B4C73DE95F05513BD09FC2E58BD2CC865CC851AD797513A9A951A3CA
03
59 0134
A6
01
-version: /
01
02
-sequence-number: /
03
03
/
58 A7
A3
01
cies: /
81
A1
01
-digest: /
82
2F
ose-alg-sha256 /
58 20                    # bytes(32)
F8690E5A86D010BF2B5348ABB99F2254DB7B608D0D626B98DB5
1AB3ECFC51907                       # unsigned(2) / suit-components
02                                # unsigned(2) / suit-components
ts: /                        # array(1)
81                                # array(1)

Tschofenig, et al. Expires 8 September 2022 [Page 58]
83                          # array(3)
4B                          # bytes(11)
544545502D446576696636D5  # "TEEP-Device"
48                          # bytes(8)
5365637572654653           # "SecureFS"
4B                          # bytes(11)
636F6E6669672E6A7366F6E    # "config.json"
04                          # unsigned(4) / suit-common-sequence: /
58 57 88                    # bytes(87)
0C                          # unsigned(8) / suit-directive-set-component-index: /
00                          # unsigned(0)
14                          # unsigned(20) / suit-directive-override-parameters: /
A4                          # map(4)
01                          # unsigned(1) / suit-parameter-vendor-identifier: /
50                          # bytes(16)
C0DDD5F15243566087DB4F5B0AA26C2F02  # unsigned(2) / suit-parameter-class-identifier: /
50                          # bytes(16)
DB42F7093D8C55BAA8C5265FC5820F4E03  # unsigned(3) / suit-parameter-image-digest: /
58 24 82 2F                 # bytes(36) / negative(15) / -16 = suit-cose-alg-sha256 /
58 20 58 24                 # bytes(32) / AAABCCDEFEEF0001223444566678889ABBCDDDDEF0
111233345567778999 0E       # unsigned(14) / suit-parameter-image-size: /
18 40                       # unsigned(64) / unsigned(1) / suit-condition-vendor-identifier: /
01                          # unsigned(64) / unsigned(1) / suit-condition-class-identifier: /
0F                          # unsigned(15) / unsigned(2) / suit-condition-image-digest: /
0F                          # unsigned(15) / unsigned(7) / suit-dependency-resolution: /
07                          # bytes(73) / array(8) / unsigned(13) / suit-directive-set-dependency-index: /
58 49 88 0D                 # unsigned(0) / unsigned(20) / suit-directive-override-parameters: /
A1                          # map(1) / unsigned(21) / suit-parameter-uri: /
78 3D                       # text(61) / 68747470733A2F2F6578616D706C652E6F72672F38643832357373612D393236642D343735342D393335332D3332646332393939376673734E73756974 # "https://exam
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ve-fetch:</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>02</td>
</tr>
<tr>
<td></td>
<td>03</td>
</tr>
<tr>
<td>n-image-match:</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>0F</td>
</tr>
<tr>
<td></td>
<td>09</td>
</tr>
<tr>
<td></td>
<td>58 2F</td>
</tr>
</tbody>
</table>

Tschofenig, et al.   Expires 8 September 2022
### E.4. Example 4: Unlink a Trusted Component

**CBOR Diagnostic Notation of SUIT Manifest**

```
8C  # array(12)
0D  # unsigned(13) / suit-directive
ve-set-dependency-index: /
  00  # unsigned(0)
  12  # unsigned(18) / suit-directive
ve-process-dependency: /
  00  # unsigned(0)
  0C  # unsigned(12) / suit-directive
ve-set-component-index: /
  00  # unsigned(0)
  14  # unsigned(20) / suit-directive
ve-override-parameters: /
  A1  # map(1)
  15  # unsigned(21) / suit-parameter-uri
  78 1F  # text(31)
  68747470733A2F2F6578616D706C652E6F72672F636F6669672E6A736F6E # "https://example.org/config.json"
  15  # unsigned(21) / suit-directive
ve-fetch: /
  02  # unsigned(2)
  03  # unsigned(3) / suit-condition
n-image-match: /
  0F  # unsigned(15)
  0A  # unsigned(10) / suit-validation
  45  # bytes(5)
  84  # unsigned(13) / suit-directive
ve-set-component-index: /
  00  # unsigned(0)
  03  # unsigned(3) / suit-condition
n-image-match: /
  0F  # unsigned(15)
```
/ SUIT_Envelope_Tagged / 107( 
  / suit-authentication-wrapper / 2: << [ 
    << [ 
      / suit-digest-algorithm-id: / -16 / suit-cose-alg-sha256 /, 
      / suit-digest-bytes: / h'632d454f19a9440a5b83493628a7ef8704c8a0205a62c34e425bAaa3C71341F42' 
    ] >>, 
    / COSE_Sign1_Tagged / 18( [ 
      / protected / << { 
        / algorithm-id / 1: -7 / ES256 / 
      } >>, 
      / unprotected: / {}, 
      / payload: / null, 
      / signature: / h'A32CDB7C1D089C27408CED3C79087220EB0D77F105BB5330912875F4D94AD108D765B650463AEB7E1CCA5084F22B2F3993176E8B3529A3202ED735E4D39BBBf' 
    ] ) >>, 
    / suit-manifest / 3: << { 
      / suit-manifest-version / 1: 1, 
      / suit-manifest-sequence-number / 2: 18446744073709551615 / UINT64_MAX /, 
      / suit-common / 3: << { 
        / suit-components / 2: [ 
          h'544545502D44657696365', / "TEEP-Device" / 
          h'536563752654653', / "SecureFS" / 
          h'8D82573A926D4754935332DC29997F74', / tc-uuid / 
          h'7461' / "ta" / 
        ], 
        / suit-common-sequence / 4: << [ 
          / suit-directive-override-parameters / 20, { 
            / suit-parameter-vendor-identifier / 1: h'C0DDD5F15243566087DB4F5B0AA26C2F', 
            / suit-parameter-class-identifier / 2: h'DB42F7093D8C55BAA8C5265FC5820F4E' 
          }, 
          / suit-condition-vendor-identifier / 1, 15, 
          / suit-condition-class-identifier / 2, 15 
        ] >>, 
        / suit-install / 9: << [ 
          / suit-directive-set-component-index: / 12, 0, 
          / suit-directive-unlink: / 33, 0 
        ] >> 
      } 
    ] 
  ] 
) 

CBOR Binary Representation
<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2F</td>
<td>Negative(15) / -16 = suit-cose-alg-sha256</td>
</tr>
<tr>
<td>82</td>
<td>Array(2)</td>
</tr>
<tr>
<td>20</td>
<td>Bytes(32)</td>
</tr>
<tr>
<td>2F</td>
<td>Negative(15) / -16 = suit-cose-alg-sha256</td>
</tr>
<tr>
<td>82</td>
<td>Array(2)</td>
</tr>
<tr>
<td>2F</td>
<td>Negative(15) / -16 = suit-cose-alg-sha256</td>
</tr>
<tr>
<td>02</td>
<td>Unsigned(2) / suit-authentication-wrapper</td>
</tr>
<tr>
<td>02</td>
<td>Unsigned(2) / suit-authentication-wrapper</td>
</tr>
<tr>
<td>1F42</td>
<td>Bytes(74)</td>
</tr>
<tr>
<td>D2</td>
<td>Tag(18) / COSE_Sign1_Tagged</td>
</tr>
<tr>
<td>43</td>
<td>Bytes(3)</td>
</tr>
<tr>
<td>A1</td>
<td>Map(1)</td>
</tr>
<tr>
<td>01</td>
<td>Unsigned(1) / algorithm-id</td>
</tr>
<tr>
<td>26</td>
<td>Negative(6) / -7 = ES256</td>
</tr>
<tr>
<td>03</td>
<td>Unsigned(3) / suit-manifest</td>
</tr>
<tr>
<td>01</td>
<td>Unsigned(1) / suit-manifest</td>
</tr>
<tr>
<td>02</td>
<td>Unsigned(2) / suit-manifest</td>
</tr>
<tr>
<td>03</td>
<td>Unsigned(3) / suit-common:</td>
</tr>
<tr>
<td>58</td>
<td>Unsigned(91)</td>
</tr>
<tr>
<td>5B</td>
<td>Map(2)</td>
</tr>
<tr>
<td>A2</td>
<td>Unsigned(2) / suit-components</td>
</tr>
<tr>
<td>81</td>
<td>Array(4)</td>
</tr>
<tr>
<td>4B</td>
<td>Bytes(11)</td>
</tr>
<tr>
<td>48</td>
<td>Bytes(8)</td>
</tr>
<tr>
<td>50</td>
<td>Bytes(16)</td>
</tr>
<tr>
<td>42</td>
<td>Bytes(2)</td>
</tr>
<tr>
<td>4D</td>
<td>&quot;ta&quot;</td>
</tr>
<tr>
<td>04</td>
<td>Unsigned(4) / suit-common-sequence:</td>
</tr>
<tr>
<td>58</td>
<td>Bytes(84)</td>
</tr>
</tbody>
</table>
86
14

ve-override-parameters:

A2

01

r-vendor-identifier:
F. Examples of SUIT Reports

This section shows some examples of SUIT reports.

F.1. Example 1: Success

SUIT Reports have no records if no conditions have failed. The URI in this example is the reference URI provided in the SUIT manifest.

```
{  
  / suit-report-manifest-digest / 1:<<[
    / algorithm-id / -16 / "sha256" /,
    / digest-bytes / h’a7fd6593ec32eb4be578278e6540c5c’
      h’09cfd7dd234973054833b2b93030609’
  ]>>,
  / suit-report-manifest-uri / 2: "tam.teep.example/personalisation.suit",
  / suit-report-records / 4: []
}
```
F.2. Example 2: Failure

{
    / suit-report-manifest-digest / 1:<<[
        / algorithm-id / -16 / "sha256" /,
        / digest-bytes / h'a7fd6593eac32eb4be578278e6540c5c09cfd7d4d234973054833b2b93030609' />
    ]>>,
    / suit-report-manifest-uri / 2: "tam.teep.example/personalisation.suit",
    / suit-report-records / 4: [
        
        {
            / suit-record-manifest-id / 1:[],
            / suit-record-manifest-section / 2: 7 / dependency-resolution /,
            / suit-record-section-offset / 3: 66,
            / suit-record-dependency-index / 5: 0,
            / suit-record-failure-reason / 6: 404
        }
    ]
}

where the dependency-resolution refers to:

107({
    authentication-wrapper,
    / manifest / 3:<<{
        / manifest-version / 1:1,
        / manifest-sequence-number / 2:3,
        common,
        dependency-resolution,
        install,
        validate,
        run,
        text
    }>>,
})

and the suit-record-section-offset refers to:

<<[
    / directive-set-dependency-index / 13,0 ,
    / directive-set-parameters / 19,{
        / uri / 21:‘tam.teep.example/’
            ‘edd94cd8-9d9c-4cc8-9216-b3ad5a2d5b8a.suit’,
    },
    / directive-fetch / 21,2 ,
    / condition-image-match / 3,15
]>>,
Authors’ Addresses

Hannes Tschofenig
Arm Ltd.
6067 Absam
Austria
Email: hannes.tschofenig@arm.com

Mingliang Pei
Broadcom
350 Ellis St
Mountain View, CA 94043
United States of America
Email: mingliang.pei@broadcom.com

David Wheeler
Amazon
United States of America
Email: dawheee@amazon.com

Dave Thaler
Microsoft
United States of America
Email: dthaler@microsoft.com

Akira Tsukamoto
AIST
Japan
Email: akira.tsukamoto@aist.go.jp
Abstract

Confidential Computing is the protection of data in use by performing computation in a hardware-based Trusted Execution Environment. Especially in virtualization environments, confidential computing could protect data and applications from access or tampering by hypervisor or other privileged users. In Computing-Aware network, computing resource is an essential element to provide computing services for network users’ applications. Introducing confidential computing in Computing-Aware network could mitigate the distrust of computing resource efficiently. This document provides the architecture of confidential computing in Computing-Aware network management plane to provide confidentiality and integrity for applications.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 2 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.
1. Introduction

The Confidential Computing Consortium defined the concept of confidential computing as "Confidential Computing is the protection of data in use by performing computation in a hardware-based Trusted Execution Environment"[CCC-White-Paper]. In detail, CPU with confidential computing feature could generate an isolated hardware-protected area, in which processing data or running code will be protected from any illegal access or tampering. In cloud computing scenario, CPU with confidential computing feature could be used to protect users’ applications and data from access or tampered by hypervisor, privileged users or other attackers in the cloud platform. In hardware industry, Intel, AMD, ARM and other chip vendors have already released their confidential computing CPU series.

In Computing-Aware network, cloud-based computing resource prepared for applications is from different places like edge or data center. If the edge or data center is outsourced or even distributed in
different security domains, not only the network administrator but also the application owner cannot trust the computing environment. The potential leakage of secret data or intellectual property will restrict the range of applications. With the protection of confidential computing, users could trust the computing environment and make sure their sensitive data and intellectual property will not be leaked.

This document introduces confidential computing to Computing-Aware network and illustrates the general architecture in network management plane. Computing-Aware network designers and users could use this document as a information reference to enhance their security.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

CC: Confidential Computing
CCR: Confidential Computing Resource
TEE: Trust Execution Environment
CCM: Confidential Computing Management
CCI: Confidential Computing Instance
TEEP: Trust Execution Environment Provisioning
TAM: Trusted Application Management

2. Motivation and Scope

2.1. Motivation

In Computing-Aware network, there is a suspicion about how to protect users’ application and data efficiently. Computing resource in Computing-Aware network is more decentralized and ambiguous than regular cloud computing. The network may distribute users’ applications in different computing platforms maintained by different administrators. If the computing platform is malicious, secret data and application intellectual property could be easily stolen or tampered. Confidential computing provides a new security model in where network users only need to trust the confidential computing hardware, firmware and the applications provided by users themselves,
any other hypervisor or software in computing platform do not have to be trusted.

2.2. Scope

This document mainly focuses on the unique features of confidential computing in network management plane. Other network planes like control/forwarding/data which have no direct interaction with confidential computing features will be ignored.


![Architecture Diagram]

Figure 1: Architecture of Confidential Computing in Computing-Aware Network

Figure 1 shows the basic architecture of confidential computing in Computing-Aware Network. This architecture refers to RATs [I-D.ietf-rats-architecture] arch and TEEP[I-D.ietf-teep-architecture] arch for remote attestation and trust execution environment provisioning. Confidential computing needs the support of CPU, in which MUST have the function of generating isolated execution environment and attesting environment. The layer of Hypervisor is for virtualization.
(1) Targeting Environment

Targeting Environment is the computing environment e.g. virtual machine, process that could provide confidentiality and integrity for applications. When used for remote attestation, the Targeting Environment will be attested by application owner. Targeting environment includes Application, Middleware, and TEEP Agent.

(2) Application

Application which runs in Computing-Aware network.

(3) Middleware

Middleware in CC has two functions: enable remote attestation and environment provisioning; provide a user-friendly environment. Some confidential computing CPU like SGX needs to use middleware to provide a environment in where applications don’t have to change their source code, e.g. Enarx [Enarx] and Occlum [Occlum].

(4) TEEP Agent

TEEP Agent is a module for provisioning middleware and application in Targeting Environment.

(5) TEEP Broker

TEEP Broker is only for communication between TEEP Agent and TAM, it doesn’t have to know any confidential information.

(6) Attesting Environment

Attesting Environment is hardware based component, like Intel Quote SGX, AMD SEV-SP, etc. This component is a part of TCB, and is used to collect targeting environment evidence for remote attestation.

(7) M/OC

M/OC is the manage and orchestration console of Computing-Aware Network.

(8) TAM

Trust Application Management, this entity is for provisioning of application and relevant middleware.

(9) Middleware Repository
This repository keeps a variety of middleware packages, which is for TAM to access based on Application type and confidential computing hardware type.

4. Environment Provisioning

When deploying applications in Computing-Aware network, TAM will choose confidential computing environment and relevant Middleware to fit their applications. Meanwhile, Computing-Aware Network needs to provide the secure procedure of provisioning middleware and applications. This document uses TEEP as reference to provision Middleware and applications in Computing-Aware network.

![Diagram](Figure 2: Application and Middleware Provisioning in Computing-Aware Network)

The Provisioning steps in Computing-Aware Network are illustrated below.

1. First, Application owner requests for confidential computing resource in Computing-Aware Network. Second, based on the request and confidential computing resource type, TAM will chose appropriate middleware.

(2) TAM establishes connections with TEEP Broker to transfer provisioning information.

(3) TEEP Broker triggers the confidential computing platform to create Targeting Environment with TEEP Agent. Then TEEP Broker establishes connections with TEEP agent. TEEP agent receives the provisioning information and unpacks it as Middleware.

Need to clarify that at this stage the Middleware doesn’t contain any secret information. The secret information of application should be provisioned after remote attestation. The specific mechanism of building targeting environment is based on specific CPU and is out of scope of this document.

5. Remote Attestation

In Computing-Aware Network, remote attestation is used for application owner to appraise if the Targeting Environment is trusted. Only after remote attestation, application owner could trust the confidential computing environment and deploy secret information. The general architecture of remote attestation in Computing-Aware Network is shown below.

![Figure 3: Remote Attestation in Computing-Aware Network](image-url)
The remote attestation steps in Computing-Aware Network are shown below. After appraising the remote attestation evidence, the application owner could deploy secret data in Targeting Environment.

1. Application owner establishes secure connection with middleware and launches remote attestation request with certain parameters like nonce.

2. Targeting Environment launches evidence collection by Middleware. Middleware sends request to Attesting Environment for remote attestation evidence. After generating evidence by Attesting Environment, the evidence will be sent back to Middleware.

3. The Application Owner requests for TEEP agent and middleware source code to generate reference value and appraise the remote attestation evidence.

4. The Targeting Environment sends the evidence to Application Owner. After appraising, Application Owner sends its application and private data to Targeting Environment.

6. Use Case

Confidential computing provides confidentiality and integrity of data and applications in the running stage. This document depicts the abstract architecture of confidential computing from the perspective of Computing-Aware Network. The following are some use cases of confidential computing in Computing-Aware Network.

VR/AR Application: Users wants to use Computing-Aware Network to host VR communication and interaction with other user. They don’t want their conversation to be awared by the network. And it is hard to encrypt all the VR context because of unacceptable cost. So, the users choose confidential computing to protect their privacy. After the remote attestation of computing environment, the users could transfer and process private information in Computing-Aware Network.

Medical Imaging Analysis: A medical institute wants to use Computing-Aware Network to share and process medical images in different branches. One primary concern is that they don’t want the patients’ medical images to be leaked. So they choose confidential computing to process these images.
7. Security Considerations

The root of trust of confidential computing is the CPU hardware. Application Owner could use the certificate or signature in remote attestation information to verify the identity of CPU. The connections between Application Owner and their applications are protected by security protocols like TLS.

8. Acknowledgements

The author would like to thank Eric Voit, Mike Bursul and Dave Thaler in CCC group who have provided valuable supports and suggestions.

9. IANA Considerations

This memo includes no request to IANA.

10. References

10.1. Normative References

[I-D.ietf-rats-architecture]

[I-D.ietf-teep-architecture]


10.2. Informative References

Yang, et al.


Authors’ Addresses

Penglin Yang
China Mobile
Email: yangpenglin@chinamobile.com

Meiling Chen
China Mobile
Email: chenmeiling@chinamobile.com

Li Su
China Mobile
Email: suli@chinamobile.com