

TEEP WG
Internet-Draft
Intended status: Informational
Expires: January 7, 2020

S. Faibish
Dell EMC
July 7, 2019

Usecases definition for IoT DDoS attacks prevention
draft-faibish-iot-ddos-usecases-00

Abstract

This document specifies several usecases related to the different ways IoT devices are exploited by malicious adversaries to instantiate Distributed Denial of Services (DDoS) attacks. The attacks are generated from IoT devices that have no proper protection against generating unsolicited communication messages targeting a certain network and creating large amounts of network traffic. The attackers take advantage of breaches in the configuration data in unprotected IoT devices exploited for DDoS attacks. The attackers take advantage of the IoT devices that can send network packets that were generated by malicious code that interacts with an OS implementation that runs on the IoT devices. The purpose of this draft is to present possible IoT DDoS usecases that need to be prevented by TEE. The major enabler of such attacks is related to IoT devices that have no OS or unprotected EE OS and run code that is downloaded to them from the TA and modified by man-in-the-middle that inserts malicious code in the OS.

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 <http://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."

The list of Internet-Draft Shadow Directories can be accessed at <https://www.ietf.org/standards/ids/internet-draft-mirror-sites/>.

This Internet-Draft will expire on January 7, 2020.

Copyright Notice

Copyright (c) 2019 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 (<http://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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	4
3. Assumptions	4
4. Usecases	4
4.1. Upgradable OS less IoT devices	4
4.2. IoT devices connected to a gateway server	5
4.3. Smart IoT devices with full OS	6
5. Security Considerations	8
9. IANA Considerations	8
10. References	8
10.1. Normative References	8
10.2. Informative References	8
Acknowledgments	9
Author's Address	9

1. Introduction

Applications executing in an IoT 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. The problem is more acute for IoT devices that run low level of OS or no OS at all and have limited ability to prevent malicious network traffic leading to DDoS. These attacks increase with the number of applications running on the device, with such other applications coming from potentially untrustworthy sources or due to man-in-the-middle mangling with the application code inserting random packets in the communication of the IoT back to operator.

The potential for attacks generated by these devices further increases with the complexity of features and applications on devices, with limited OS capabilities, running code that is downloaded from untrustworthy operators. The danger of attacks on a OS less system increases as the data transmitted by the devices to the operator increases.

As an example, an IoT device that sends pollution data each minute from city wide sensors to a cloud application that analyses city air quality and generate reports and warning to the public can be used to send random data at much higher frequency like 1000 per second. This malicious transmission can shut down the cloud receiving this data. The worst part of this is that the IoT device OS has no idea that the transmission is wrong and is creating DDoS for the cloud used by the IoT devices. Additionally there could be coordinated attacks coming from many IoT devices connected to the same cloud and shut down all the cloud services.

In general case there is an edge server to which the IoT devices are connected and the server is managing the management of the data transmitted to the OA. In this case the edge server has an OS and a TEE that can prevent DDoS attacks that were generated by the IoT devices if the transmission is malicious. Moreover the edge server will facilitate the code upgrade and prevent malicious code being stored on the device code. So, the edge server will become the TEE for all the devices connected to it. Moreover if the code of the device is compromised the edge server will block the packets that were generated by the IoT devices connected to it.

According to analysts study DDoS originated from IoT devices accounted for 90% of all the DDoS attacks and increased 10x in 2018 ([1]) and the majority of the attacks were from devices with limited compute and OS resources as well as webcams with REE. This will require special TEE protocol support preventing the use of these devices for DDoS attacks. This draft is trying to present the usecases that enable such attacks with the intention to request that TEEP WG addresses this special security loophole. And the major problem resides in the inability of IoT devices to prevent broadcasting network packets generated by unauthorized code, inserted at upgrade time, to execute on devices with low compute capabilities.

Trusted Execution Environments (TEEs), including Intel SGX, ARM TrustZone, Secure Elements, and others, can enforce that only authorized code can execute within the TEE, and any memory used by such code is protected against tampering or disclosure outside the TEE. This observation is only true if there is awareness that IoT devices are enabled to send data back to the cloud and or the SP that did the upgrade. In such environments malicious code includes a method of external triggered or time based attacks.

In most such devices there is none or limited "Trusted Agent" or "Trusted Application Manager (TAM)" on the client side running inside the TEE. The purpose of this draft is to present 3 DDoS usecases that TEEP needs to address prevention of using the IoT devices as the origin of such attacks.

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-tee-architecture], including Trusted Execution Environment (TEE), Trusted Application (TA), Trusted Application Manager (TAM), Agent, and Broker.

3. Assumptions

This draft assumes that an applicable device may or may not be equipped with any TEEs nor pre-provisioned with a device-unique public/private key pair, which is securely stored.

A TEE uses an isolation mechanism between Trusted Applications to ensure that one TA cannot read, modify or delete the data and code of another TA. We also assume that there can be a TEE running in a edge server to which the devices may be connected. The edge server will include such a TEE and will become the secure gateway as client/agent.

4. Usecases

4.1 Upgradable OS less IoT devices

The simplest IoT device we refer to here is a device that has enough OS and EE to perform a single function like sending back to the broker time series at given time intervals, Figure 1. As an example an IoT device that monitors the air quality in a city and send back to the cloud this data that will be aggregated with many sensors around the city. The device will run simple code that can be executed on the device and at a minimum it will be capable to receive and install code upgrades from the Broker. Such devices have very limited or no security or trust protection and it can be exposed to man-in-the-middle attacks target by malicious actors that are trying to insert malicious code MA (Malicious Application) in the upgraded code.

One example of such code may include a trigger, that can be activated in a similar manner as the code upgrade request, and used to start DDoS attacks coordinated as a cluster. As the device function is to send time series data to the cloud the malicious code can send same data 1000s of times flooding the recipient cloud from all the devices in the cluster. As a second example the malicious code can use a timer and start sending empty network packets back to the provider network and also targeting a given IP address of a victim. Such examples are attacks related to Mirai botnets also in [2] as similar attacks were uncovered targeting for example financial institutions in order to hide cyberattacks for stilling money or even crypto-currency.

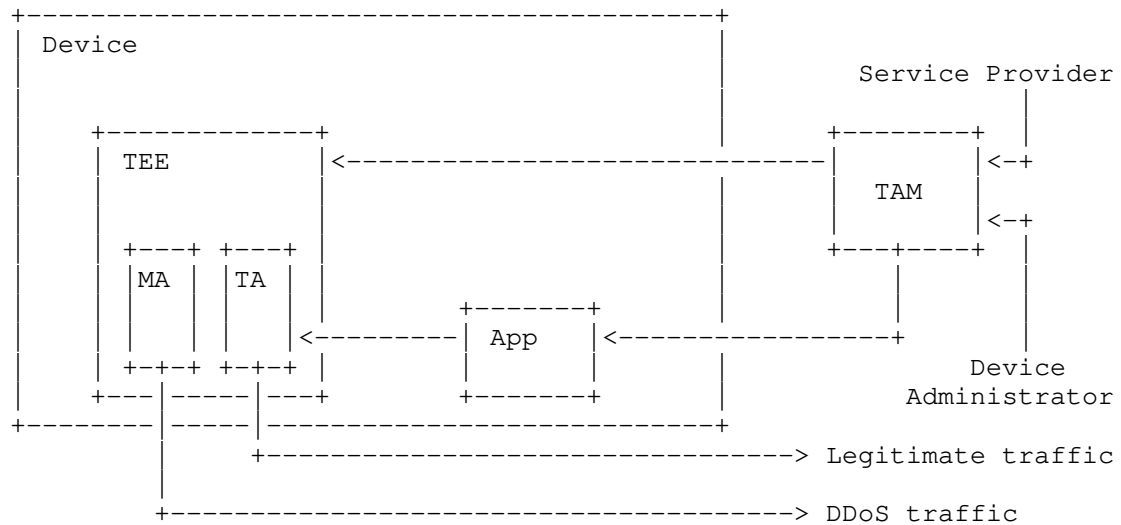


Figure 1: OS less IoT devices diagram

4.2 IoT devices connected to a gateway server

In this case the OS less IoT device is connected to a local edge TEEP server which has rich execution OS and acts as a bridge between the device and the cloud collecting the time series from the device. In this usecase the upgrades are done via the edge gateway server that has full TEEP capabilities and can detect DDoS attacks and prevent the DDoS traffic to escape outside the edge server. For example the edge gateway server could be a computer with full security protection or a mobile device such as a tablet or even a cell phone. We assume that in this case the edge server has a full TEE and can manage several IoT devices running multiple different applications. We also assume that the edge server is connected to a TEEP Broker. For example webcams can be such IoT devices connected to gateway server used for DDoS attacks [1].

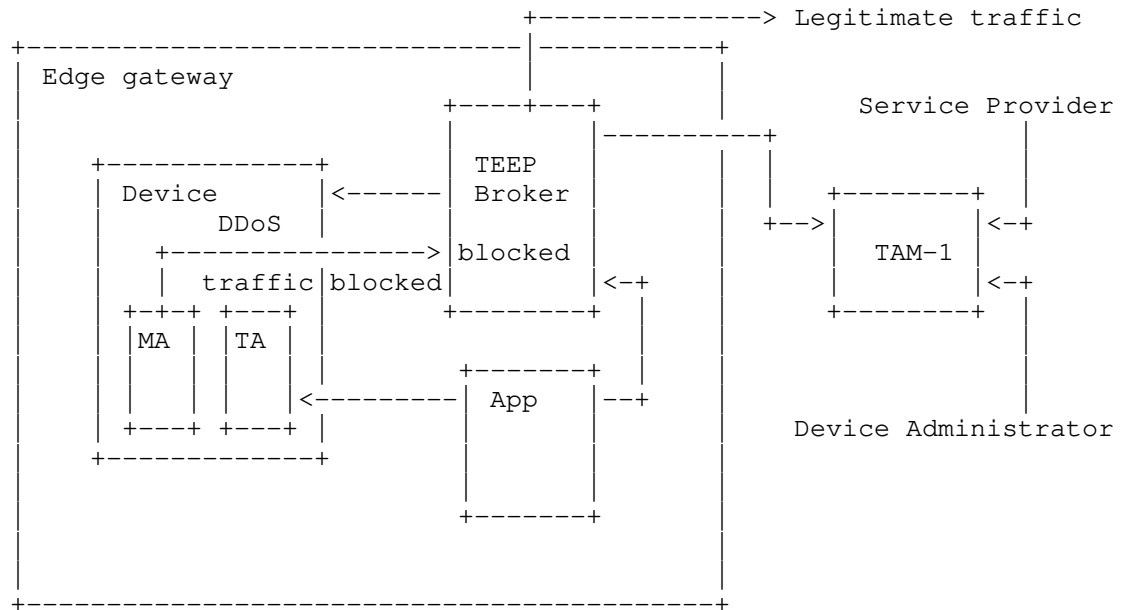


Figure 2: OS less IoT devices connected to gateway server

In this scenario the DDoS traffic is only generating network traffic inside the edge limits and can be stopped by the TEE inside the server. For example when an edge server is connected to home appliances such as home temperature control or electricity and water meters that are supposed to send time series to the cloud, triggering a DDoS will not be allowed to send packets outside the gateway limits. It can still prevent sending the sensing data to the cloud destination but TEEP will prevent DDoS traffic outside the edge server. Additional to this using TEEP will prevent code upgrades done from untrusted sources and even detect malicious code to install on the device. In this configuration (Figure 2) SPs do not directly interact with devices. DAs may elect to use a TAM for remote administration of TAs instead of managing each device directly. Moreover the Legitimate traffic can be sanitized to prevent malicious code spread to other devices.

4.3 Smart IoT devices with full OS

The Internet of Things (IoT) has been posing threats to networks and national infrastructures because of existing weak security in devices. But there are IoT devices and systems that have the OS and compute power to detect and prevent malware for generation DDoS attacks. It is possible that for such devices can implement measures to prevent malware from manipulating actuators (e.g., IoT controlling computer assisted automobiles or self driving cars), or forcing such cars into accidents and damage infrastructures and even lose life.

Such an experiment was done in the research communities and there was even a contest about how fast hackers can take control of a car using automatic driving. The results were that the current security of such cars is not strong enough to prevent taking control over the internet. A TEE can be the best way to implement such IoT security functions for "smart" environments using advanced OSes such as cars.

TEEs could be used to store variety of sensitive data for IoT devices. For example, a TEE could be used in smart cars to store a driver's biometric information for identification, available in some new cars, and for protecting access driving wheel control mechanism. Figure 3 presents the architecture of such a self driving car. In this usecase the applications run inside the TEE and are connected to the service provider's cloud similar to some "connected" cars (BMW for example). The applications running inside the TEE can be either monitoring functions or car status (TA1) or diagnostic malfunctions (TA2). All these applications can be vital to the operation of the car and the safety of the drivers and roads. In general in this usecase the Service provider and the Device Administrator are represented by the vehicle manufacturer during the warranty time and after that they can be a different service provider doing maintenance.

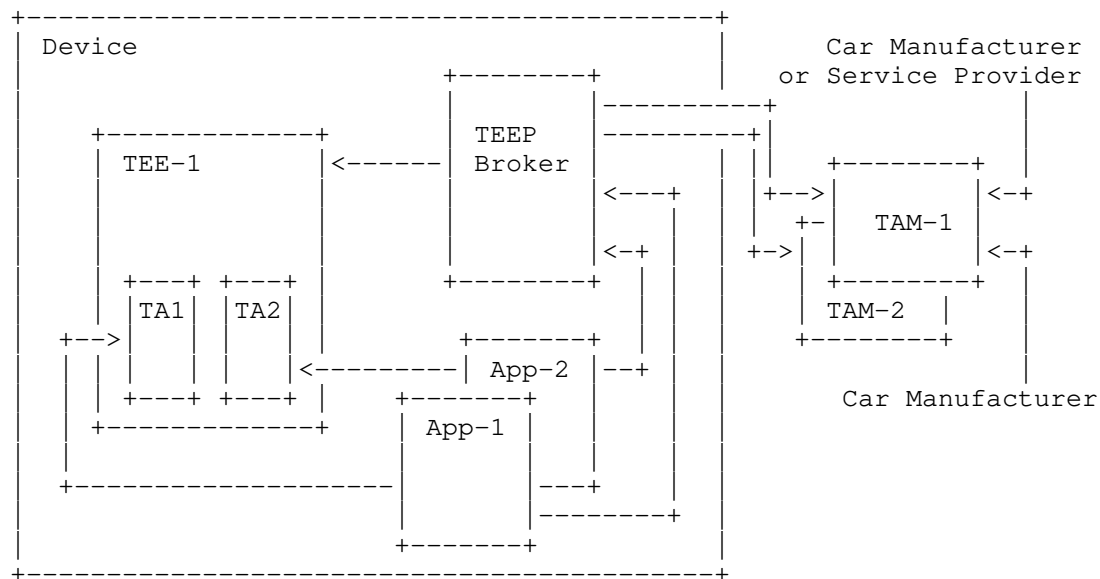


Figure 3: OS capable IoT devices for connected cars

There are additional usecases similar to this one like electric power and grid monitoring and control that have rich compute and memory resources running in a centralized location (secure) or in the cloud (unsecure) but need high levels of security.

There are additional security models of IoT devices that can fit in these 3 examples and we will extend the protocols to apply to as many as we can consider as useful.

8. Security Considerations

Although TEEP architecture document [I-D.ietf-teep-architecture] addresses some IoT devices examples there are IoT usecases that require more detailed design and better definitions of the Broker behavior in different usecases discussed in this draft. As such, Broker implementations MUST support many of this usecases critical for security and safety.

9. IANA Considerations

This document does not require actions by IANA.

10. References

10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

10.2. Informative References

- [1] Vljajic, N., and Zhou, D., "IoT as a Land of Opportunity for DDoS Hackers", Computer Magazine, July 2018, pp. 26-34.
- [2] Kolias, C., Kambourakis, G., Stavrou, A., and Voas, J., "DDoS in the IoT: Mirai and Other Botnets", Computer Magazine, July 2017, pp. 80-84.
- [I-D.ietf-teep-architecture] Pei, M., Tschofenig, H., Wheeler, D., Atyeo, A., and D. Liu, "Trusted Execution Environment Provisioning (TEEP) Architecture", draft-ietf-teep-architecture-02 (work in progress), March 2019.
- [GPTEE] Global Platform, "GlobalPlatform Device Technology: TEE System Architecture, v1.1", Global Platform GPD_SPE_009, January 2017, <<https://globalplatform.org/specs-library/tee-system-architecture-v1-1/>>.

Acknowledgments

This draft has attempted to capture many IoT security usecases known to the author and presented in the literature as well as discussed in the security forums. These usecases present challenges both for DDoS attacks that became critical as well as applied security for new autonomous devices. We proposed to add these usecases to the TEEP Architecture draft.

Author's Address

Sorin Faibish
Dell EMC
228 South Street
Hopkinton, MA 01774
United States of America

Phone: +1 508-249-5745
Email: faibish.sorin@dell.com

TEEP WG
Internet-Draft
Intended status: Informational
Expires: June 19, 2022

S. Faibish
Cirrus Data Solutions Inc.
M. K. Chowdhury
Deloitte Canada
December 19, 2021

Test Tools for IoT DDoS vulnerability scanning
draft-faibish-iot-ddos-usecases-06

Abstract

This document specifies several usecases related to the different ways IoT devices are exploited by malicious adversaries to instantiate Distributed Denial of Services (DDoS) attacks. The attacks are generated from IoT devices that have no proper protection against generating unsolicited communication messages targeting a certain network and creating large amounts of network traffic. The attackers take advantage of breaches in the configuration data in unprotected IoT devices exploited for DDoS attacks. The attackers take advantage of the IoT devices that can send network packets that were generated by malicious code that interacts with an OS implementation that runs on the IoT devices. The purpose of this draft is to present possible IoT DDoS usecases that need to be prevented by TEE. The major enabler of such attacks is related to IoT devices that have no OS or unprotected EE OS and run code that is downloaded to them from the TA and modified by man-in-the-middle that inserts malicious code in the OS. This draft adds list of MUD files for most IoT devices.

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 <http://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."

The list of Internet-Draft Shadow Directories can be accessed at <https://www.ietf.org/standards/ids/internet-draft-mirror-sites/>.

This Internet-Draft will expire on June 19, 2022.

Copyright Notice

Copyright (c) 2021 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 (<http://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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	4
3. Assumptions	4
4. Usecases	5
4.1. Upgradable OS less IoT devices	5
4.2. IoT devices connected to a gateway server	6
4.3. Smart IoT devices with full OS	7
5. Security Considerations	8
6. IANA Considerations	8
7. References	8
7.1. Normative References	8
7.2. Informative References	9
Acknowledgments	9
Author's Address	10

1. Introduction

Problems with IoT devices arise from the fact that manufacturers ship their devices with almost no security measures and the companies that buy these IoT devices don't have proper visibility/understanding of their networks with these new products. Applications executing in an IoT 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. The problem is more acute for IoT devices that run low level of OS or no OS at all and have limited ability to prevent malicious network traffic leading to DDoS. These attacks increase with the number of applications running on the device, with such other applications coming from potentially untrustworthy sources or due to man-in-the-middle mangling with the application code inserting random packets in the communication of the IoT back to operator.

The potential for attacks generated by these devices further increases with the complexity of features and applications on devices, with limited OS capabilities, running code that is downloaded from untrustworthy operators.

The danger of attacks on an OS-less system increases as the data transmitted by the devices to the operator increases. There is provision in the MUD protocol [RFC8520] to add security measures but it does not replace other security measures and only complement existing security measures if they are already in place.

But for MUD to work, every IoT device requires a unique MUD-URL specific to the kind of device type/model and matches the needed configuration manifest. There is a MUD file that SHOULD be provided by the device manufacturing that contains instructions for the expected behavior of the device. For cheap OS-less devices the manufacturer does not always provide the accurate behavior and require the network routers to detect malicious traffic and stop it. Abnormal behavior could be limiting the peak request rate of how many requests per minute is normal for the device that is used as prevention control of DDoS. Unfortunately IoT devices manufacturers do not always generate MUD profiles specific to their devices or even their companies. MUD in itself is an important step towards securing IoT devices but it is not enough for preventing DDoS attacks.

As an example, an IoT device that sends pollution data each minute from city wide sensors to a cloud application that analyses city air quality and generate reports and warning to the public can be used to send random data at much higher frequency like 1000 per second. This malicious transmission can shut down the cloud receiving this data. The worst part of this is that the IoT device OS has no idea that the transmission is wrong and is creating DDoS for the cloud used by the IoT devices. Additionally there could be coordinated attacks coming from many IoT devices connected to the same cloud and shut down all the cloud services.

In general case there is an edge server to which the IoT devices are connected and the server is managing the management of the data transmitted to the OA. In this case the edge server has an OS and a TEE that can prevent DDoS attacks that were generated by the IoT devices if the transmission is malicious. Moreover the edge server will facilitate the code upgrade and prevent malicious code being stored on the device code. So, the edge server will become the TEE for all the devices connected to it. Moreover if the code of the device is compromised the edge server will block the packets that were generated by the IoT devices connected to it.

According to analysts study DDoS originated from IoT devices accounted for 90% of all the DDoS attacks and increased 10x in 2018 ([1]) and the majority of the attacks were from devices with limited compute and OS resources as well as webcams with REE.

This will require special TEE protocol support preventing the use of these devices for DDoS attacks. This draft is trying to present the usecases that enable such attacks with the intention to request that TEEP WG addresses this special security loophole. And the major problem resides in the inability of IoT devices to prevent broadcasting network packets generated by unauthorized code, inserted at upgrade time, to execute on devices with low compute capabilities.

Trusted Execution Environments (TEEs), including Intel SGX, ARM TrustZone, Secure Elements, and others, can enforce that only authorized code can execute within the TEE, and any memory used by such code is protected against tampering or disclosure outside the TEE. This observation is only true if there is awareness that IoT devices are enabled to send data back to the cloud and or the SP that did the upgrade. In such environments malicious code includes a method of external triggered or time based attacks.

In most such devices there is none or limited "Trusted Agent" or "Trusted Application Manager (TAM)" on the client side running inside the TEE. The purpose of this draft is to present 3 DDoS usecases that TEEP needs to address prevention of using the IoT devices as the origin of such attacks.

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), Agent, and Broker.

3. Assumptions

This draft assumes that an applicable device may or may not be equipped with any TEEs nor pre-provisioned with a device-unique public/private key pair, which is securely stored.

A TEE uses an isolation mechanism between Trusted Applications to ensure that one TA cannot read, modify or delete the data and code of another TA. We also assume that there can be a TEE running in a edge server to which the devices may be connected. The edge server will include such a TEE and will become the secure gateway as client/agent.

4. Usecases

4.1 Upgradable OS less IoT devices

The simplest IoT device we refer to here is a device that has enough OS and EE to perform a single function like sending back to the broker time series at given time intervals, Figure 1. As an example an IoT device that monitors the air quality in a city and send back to the cloud this data that will be aggregated with many sensors around the city. The device will run simple code that can be executed on the device and at a minimum it will be capable to receive and install code upgrades from the Broker. Such devices have very limited or no security or trust protection and it can be exposed to man-in-the-middle attacks target by malicious actors that are trying to insert malicious code MA (Malicious Application) in the upgraded code.

One example of such code may include a trigger, that can be activated in a similar manner as the code upgrade request, and used to start DDoS attacks coordinated as a cluster. As the device function is to send time series data to the cloud the malicious code can send same data 1000s of times fludding the recipient cloud from all the devices in the cluster. As a second example the malicious code can use a timer and start sending empty network packets back to the provider network and also targeting a given IP address of a victim. Such examples are attacks related to Mirai botnets also in [2] as similar attacks were uncovered tergeting for example financial institutions in order to hide cyberattacks for stilling money or even crypto-currency.

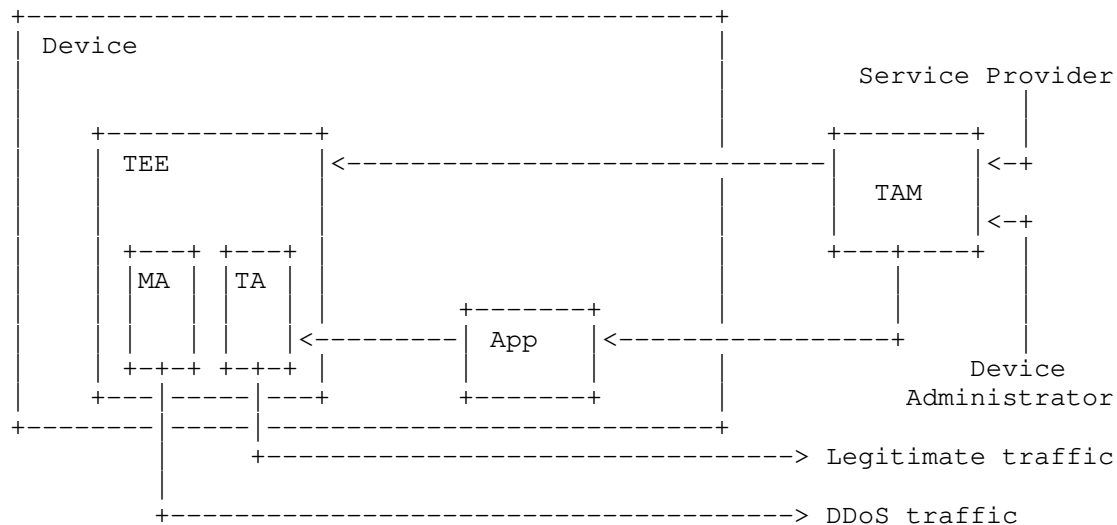


Figure 1: OS less IoT devices diagram

4.2 IoT devices connected to a gateway server

In this case the OS less IoT device is connected to a local edge TEEP server which has rich execution OS and acts as a bridge between the device and the cloud collecting the time series from the device. In this usecase the upgrades are done via the edge gateway server that has full TEEP capabilities and can detect DDoS attacks and prevent the DDoS traffic to escape outside the edge server. For example the edge gateway server could be a computer with full security protection or a mobile device such as a tablet or even a cell phone. We assume that in this case the edge server has a full TEE and can manage several IoT devices running multiple different applications. We also assume that the edge server is connected to a TEEP Broker. For example webcams can be such IoT devices connected to gateway server used for DDoS attacks [1] spoofing [4]. A list of such IoT devices MUD files is found in [3] used by python tool [5] to test vulnerabilities is available.

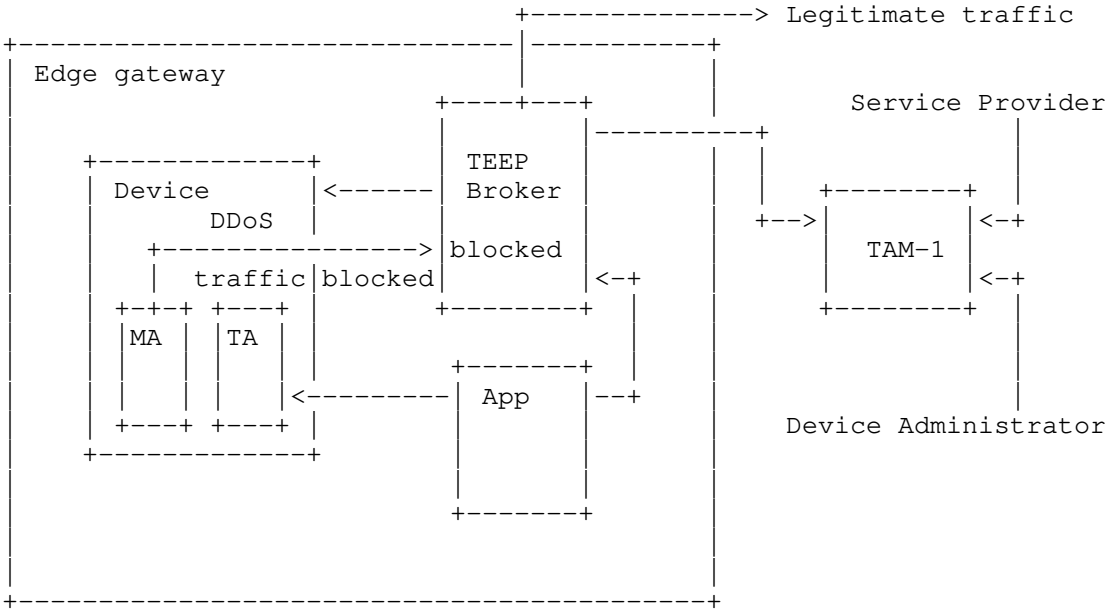


Figure 2: OS less IoT devices connected to gateway server

In this scenario the DDoS traffic is only generating network traffic inside the edge limits and can be stopped by the TEE inside the server. For example when an edge server is connected to home appliances such as home temperature control or electricity and water meters that are supposed to send time series to the cloud, triggering a DDoS will not be allowed to send packets outside the gateway limits.

It can still prevent sending the sensing data to the cloud destination but TEEP will prevent DDoS traffic outside the edge server. Additional to this using TEEP will prevent code upgrades done from untrusted sources and even detect malicious code to install on the device. In this configuration (Figure 2) SPs do not directly interact with devices. DAs may elect to use a TAM for remote administration of TAs instead of managing each device directly. Moreover the Legitimate traffic can be sanitized to prevent malicious code spread to other devices.

4.3 Smart IoT devices with full OS

The Internet of Things (IoT) has been posing threats to networks and national infrastructures because of existing weak security in devices. But there are IoT devices and systems that have the OS and compute power to detect and prevent malware for generation DDoS attacks. It is possible that for such devices can implement measures to prevent malware from manipulating actuators (e.g., IoT controlling computer assisted automobiles or self driving cars), or forcing such cars into accidents and damage infrastructures and even lose life.

Such an experiment was done in the research communities and there was even a contest about how fast hackers can take control of a car using automatic driving. The results were that the current security of such cars is not strong enough to prevent taking control over the internet. A TEE can be the best way to implement such IoT security functions for "smart" environments using advanced Oses such as cars.

TEEs could be used to store variety of sensitive data for IoT devices. For example, a TEE could be used in smart cars to store a driver's biometric information for identification, available in some new cars, and for protecting access driving wheel control mechanism. Figure 3 presents the architecture of such a self driving car. In this usecase the applications run inside the TEE and are connected to the service provider's cloud similar to some "connected" cars (BMW for example). The applications running inside the TEE can be either monitoring functions or car status (TA1) or diagnostic malfunctions (TA2). All these applications can be vital to the operation of the car and the safety of the drivers and roads. In general in this usecase the Service provider and the Device Administrator are represented by the vehicle manufacturer during the warranty time and after that they can be a different service provider doing maintenance.

There are additional usecases similar to this one like electric power and grid monitoring and control that have rich compute and memory resources running in a centralized location (secure) or in the cloud (unsecure) but need high levels of security.

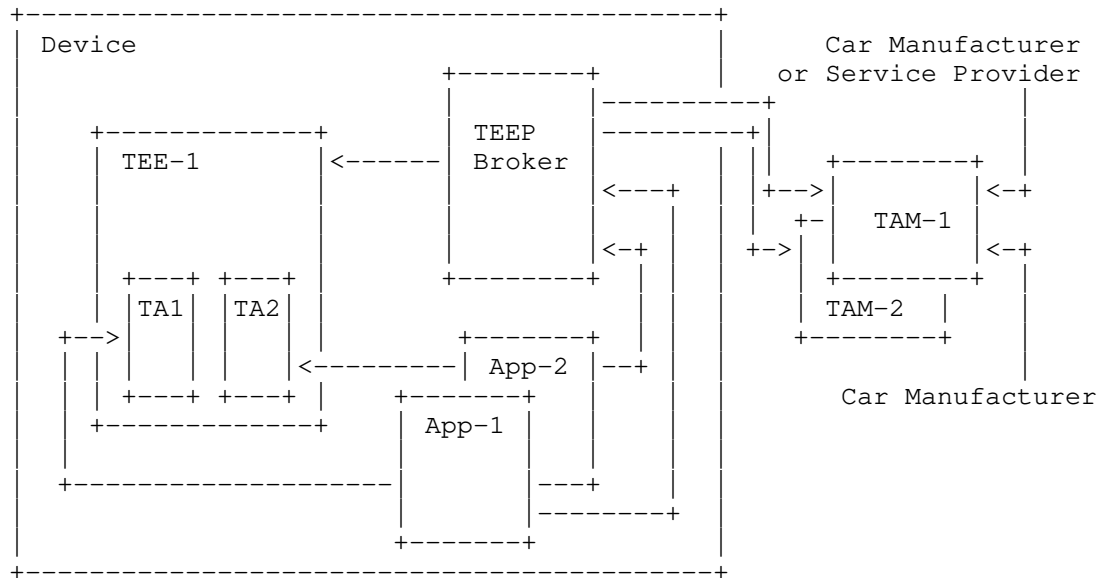


Figure 3: OS capable IoT devices for connected cars

There are additional security models of IoT devices that can fit in these 3 examples and we will extend the protocols to apply to as many as we can consider as useful.

5. Security Considerations

Although TEEP architecture document [I-D.ietf-teep-architecture] addresses some IoT devices examples there are IoT usecases that require more detailed design and better definitions of the Broker behavior in different usecases discussed in this draft. As such, Broker implementations MUST support many of this usecases critical for security and safety.

6. IANA Considerations

This document does not require actions by IANA.

7. References

7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8520] Lear, E., Droms, R. and Romascanu, D., "Manufacturer Usage Description Specification", March 2019, <<https://www.rfc-editor.org/rfc/rfc8520.txt>>

7.2. Informative References

- [1] Vljajic, N., and Zhou, D., "IoT as a Land of Opportunity for DDoS Hackers", Computer Magazine, July 2018, pp. 26-34.
- [2] Kolias, C., Kambourakis, G., Stavrou, A., and Voas, J., IP Spoofing In and Out of the Public Cloud: From Policy to Practice,
- [3] MUD files for testing DDoS vulnerabilities of IoT devices, <https://iotanalytics.unsw.edu.au/mudprofiles/>
- [4] Vljajic N., Chowdhury M., and Marin Litoiu, "IP Spoofing In and Out of the Public Cloud: From Policy to Practice", Computers 2019, 8(4), 81; <https://doi.org/10.3390/computers8040081>
- [5] IoT devices scanner for DDoS vulnerabilities test tool, python code, <https://github.com/mashrufkabir/IoTScanner>
- [I-D.ietf-teep-architecture] Pei, M., Tschofenig H., Thaler, D., Wheeler, D., "Trusted Execution Environment Provisioning (TEEP) Architecture", draft-ietf-teep-architecture-15 (work in progress), July 2021.

Acknowledgments

This draft has attempted to capture many IoT security usecases known to the author and presented in the literature as well as discussed in the security forums. These usecases present challenges both for DDoS attacks that became critical as well as applied security for new autonomous devices. We proposed to add these usecases to the TEEP Architecture draft.

Author's Address

Sorin Faibish
Cirrus Data Solutions Inc.
11 Selwyn Road
Newton, MA 02461
United States of America

Phone: +1 617-510-0422
Email: sorin.faibish@cdsi.us.com

Mashruf Kabir Chowdhury
Deloitte Canada
Apt 2112 - 70 Temperance St
Toronto, Ontario M5H 0B1
Canada

Phone: +1-416-388-5146
Email: mashrufkabir@icloud.com

TEEP
Internet-Draft
Intended status: Informational
Expires: January 9, 2020

M. Pei
Symantec
H. Tschofenig
Arm Limited
D. Wheeler
Intel
A. Atyeo
Intercede
L. Dapeng
Alibaba Group
July 08, 2019

Trusted Execution Environment Provisioning (TEEP) Architecture
draft-ietf-teep-architecture-03

Abstract

A Trusted Execution Environment (TEE) is designed to provide a hardware-isolation mechanism to separate a regular operating system from security-sensitive application components.

This architecture document motivates the design and standardization of a protocol for managing the lifecycle of trusted applications running 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 January 9, 2020.

Copyright Notice

Copyright (c) 2019 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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	3
2. Terminology	5
3. Assumptions	8
4. Use Cases	8
4.1. Payment	8
4.2. Authentication	9
4.3. Internet of Things	9
4.4. Confidential Cloud Computing	9
5. Architecture	9
5.1. System Components	9
5.2. Different Renditions of TEEP Architecture	12
5.3. Multiple TAMs and Relationship to TAs	14
5.4. Client Apps, Trusted Apps, and Personalization Data	15
5.5. Examples of Application Delivery Mechanisms in Existing TEEs	16
5.6. TEEP Architectural Support for Client App, TA, and Personalization Data Delivery	17
5.7. Entity Relations	17
5.8. Trust Anchors in TEE	20
5.9. Trust Anchors in TAM	20
5.10. Keys and Certificate Types	21
5.11. Scalability	23
5.12. Message Security	23
5.13. Security Domain	23

5.14. A Sample Device Setup Flow	23
6. TEEP Broker	24
6.1. Role of the TEEP Broker	25
6.2. TEEP Broker Implementation Consideration	25
6.2.1. TEEP Broker Distribution	26
6.2.2. Number of TEEP Brokers	26
7. Attestation	26
7.1. Attestation Cryptographic Properties	28
7.2. TEEP Attestation Structure	29
7.3. TEEP Attestation Claims	31
7.4. TEEP Attestation Flow	31
7.5. Attestation Key Example	31
7.5.1. Attestation Hierarchy Establishment: Manufacture	32
7.5.2. Attestation Hierarchy Establishment: Device Boot	32
7.5.3. Attestation Hierarchy Establishment: TAM	32
8. Algorithm and Attestation Agility	32
9. Security Considerations	33
9.1. TA Trust Check at TEE	33
9.2. One TA Multiple SP Case	33
9.3. Broker Trust Model	34
9.4. Data Protection at TAM and TEE	34
9.5. Compromised CA	34
9.6. Compromised TAM	34
9.7. Certificate Renewal	34
10. IANA Considerations	35
11. Acknowledgements	35
12. References	35
12.1. Normative References	35
12.2. Informative References	35
Appendix A. History	37
Authors' Addresses	37

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 increase 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 execute applications in a protected environment that separates applications inside the TEE from the regular operating system and from other applications on the device. 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 a normal application running in the regular operating system is referred to as an Untrusted Application (UA).

The TEE uses hardware to enforce protections on the TA and its data, but also presents a more limited set of services to applications inside the TEE than is normally available to UA's running in the normal operating system.

But not all TEEs are the same, 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 developers and service providers interacting with TAs in a TEE, an interoperable protocol for managing TAs running in different TEEs of various devices is needed. In this TEE ecosystem, there often arises a need for an external trusted party to verify the identity, claims, and rights of Service Providers (SP), devices, and their TEEs. This trusted third party is the Trusted Application Manager (TAM).

This protocol addresses the following problems:

- A Service Provider (SP) intending to provide services through a TA to users of a device needs to determine security-relevant information of a device before provisioning their TA to the TEE within the device. Examples include the verification of the device 'root of trust' and the type of TEE included in a device.
- A TEE in a device needs to determine whether a Service Provider (SP) that wants to manage a TA in the device is authorized to manage TAs in the TEE, and what TAs the SP is permitted to manage.

- The parties involved in the protocol must be able to attest that a TEE is genuine and capable of providing the security protections required by a particular TA.
- A Service Provider (SP) must be able to determine if a TA exists (is installed) on a device (in the TEE), and if not, install the TA in the TEE.
- A Service Provider (SP) must be able 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 Service Provider (SP) must be able to remove a TA in a device's TEE if the SP is no longer offering such services or the services are being revoked from a particular user (or device). For example, if a subscription or contract for a particular service has expired, or a payment by the user has not been completed or has been rescinded.
- A Service Provider (SP) must be able to define the relationship between cooperating TAs under the SP's control, and specify whether the TAs can communicate, share data, and/or share key material.

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.

The following terms are used:

- Client Application: An application running in a Rich Execution Environment, such as an Android, Windows, or iOS application. We sometimes refer to this as the 'Client App'.
- Device: A physical piece of hardware that hosts a TEE along with a Rich Execution Environment. A Device contains a default list of Trust Anchors that identify entities (e.g., TAMs) that are trusted by the Device. This list is normally set by the Device Manufacturer, and may be governed by the Device's network carrier. The list of Trust Anchors is normally modifiable by the Device's owner or Device Administrator. However the Device manufacturer and network carrier may restrict some modifications, for example, by not allowing the manufacturer or carrier's Trust Anchor to be removed or disabled.

- 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. This environment and applications running on it are considered un-trusted.
- Service Provider (SP): An entity that wishes to provide a service on Devices that requires the use of one or more Trusted Applications. A Service Provider requires the help of a TAM in order to provision the Trusted Applications to remote devices.
- 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., parent allowing children to use their tablet or laptop). Relates to Device Owner and Device Administrator.
- Device Owner: A device is always owned by someone. 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 device users have no or limited administration rights. In this case, the enterprise appoints a device administrator that is not the device owner.
- Device Administrator (DA): 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 applications and TAs, approve or reject Trust Anchors, and approve or reject Service Providers, 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 administrate a device through a TAM.
- Trust Anchor: A public key in a device whose corresponding private key is held by an entity implicitly trusted by the device. The Trust Anchor may be a certificate or it may be a raw public key along with additional data if necessary such as its public key algorithm and parameters. The Trust Anchor is normally stored in a location that resists unauthorized modification, insertion, or replacement. The digital fingerprint of a Trust Anchor may be stored along with the Trust Anchor certificate or public key. A device can use the fingerprint to uniquely identify a Trust Anchor. The Trust Anchor private key owner can sign certificates of other public keys, which conveys trust about those keys to the

device. A certificate signed by the Trust Anchor communicates that the private key holder of the signed certificate is trusted by the Trust Anchor holder, and can therefore be trusted by the device. Trust Anchors in a device may be updated by an authorized party when a Trust Anchor should be deprecated or a new Trust Anchor should be added.

- Trusted Application (TA): An application component that runs in a TEE.
- Trusted Execution Environment (TEE): An execution environment that runs alongside of, but is isolated from, an REE. A TEE has security capabilities and meets certain security-related requirements. It protects TEE assets from general software attacks, defines rigid safeguards as to data and functions that a program can access, and resists a set of defined threats. It should have at least the following three properties:
 - (a) A device unique credential that cannot be cloned;
 - (b) Assurance that only authorized code can run in the TEE;
 - (c) Memory that cannot be read by code outside the TEE.

There are multiple technologies that can be used to implement a TEE, and the level of security achieved varies accordingly.

- Root-of-Trust (RoT): A hardware or software component in a device that is inherently trusted to perform a certain security-critical function. A RoT should be secure by design, small, and protected by hardware against modification or interference. Examples of RoTs include software/firmware measurement and verification using a Trust Anchor (RoT for Verification), provide signed assertions using a protected attestation key (RoT for Reporting), or protect the storage and/or use of cryptographic keys (RoT for Storage). Other RoTs are possible, including RoT for Integrity, and RoT for Measurement. Reference: NIST SP800-164 (Draft).
- Trusted Firmware (TFW): A firmware in a device that can be verified with a Trust Anchor by RoT for Verification.
- Bootloader key: This symmetric key is protected by electronic fuse (eFUSE) technology. In this context it is used to decrypt a TFW private key, which belongs to a device-unique private/public key pair. Not every device is equipped with a bootloader key.

This document uses the following abbreviations:

- CA: Certificate Authority
- REE: Rich Execution Environment
- RoT: Root of Trust
- SD: Security Domain
- SP: Service Provider
- TA: Trusted Application
- TAM: Trusted Application Manager
- TEE: Trusted Execution Environment
- TFW: Trusted Firmware

3. Assumptions

This specification assumes that an applicable device is equipped with one or more TEEs and each TEE is pre-provisioned with a device-unique public/private key pair, which is securely stored.

A TEE uses an isolation mechanism between Trusted Applications to ensure that one TA cannot read, modify or delete the data and code of another TA.

4. Use Cases

4.1. Payment

A payment application in a mobile device requires high security and trust about 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 authentication.

A secure user interface (UI) may be used in a mobile device to prevent malicious software from stealing sensitive user input data. Such an application implementation often relies on a TEE for user input protection.

4.2. Authentication

For better security of authentication, a device may store its sensitive authentication keys inside a TEE, providing hardware-protected security key strength and trusted code execution.

4.3. Internet of Things

The Internet of Things (IoT) has been posing threats to networks and national infrastructures because of existing weak security in devices. It is very 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.

TEEs could be used to store variety of sensitive data for IoT devices. For example, a TEE could be used in smart door locks to store a user's biometric information for identification, and for protecting access the locking mechanism.

4.4. Confidential Cloud Computing

A tenant can store sensitive data 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 provider for reduced liability and increased cloud adoption.

5. Architecture

5.1. System Components

The following are the main components in the system. Full descriptions of components not previously defined are provided below. Interactions of all components are further explained in the following paragraphs.

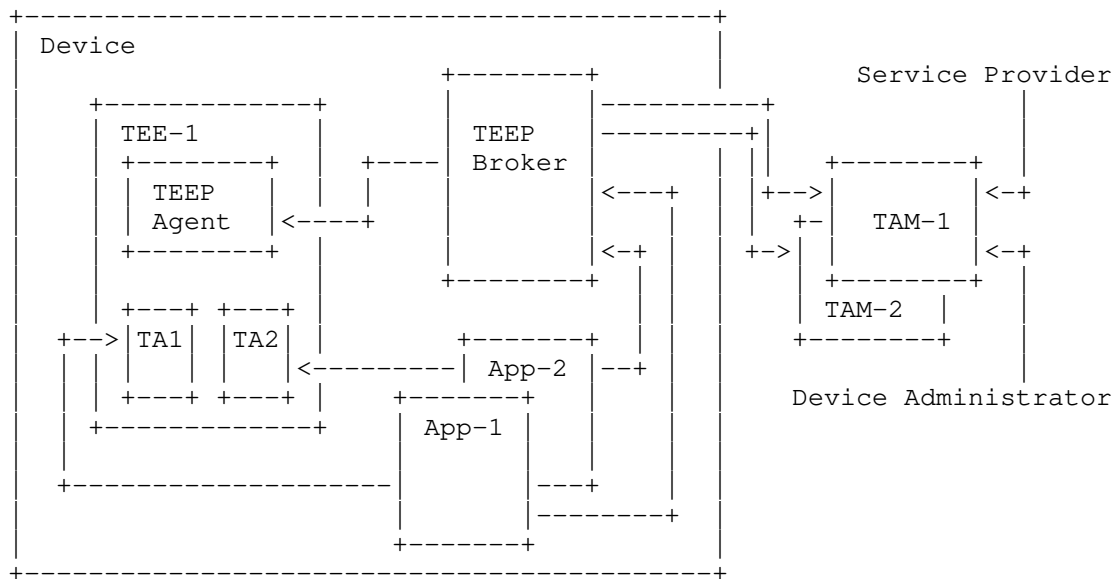


Figure 1: Notional Architecture of TEEP

- Service Providers (SP) and Device Administrators (DA) utilize the services of a TAM to manage TAs on Devices. SPs do not directly interact with devices. DAs may elect to use a TAM for remote administration of TAs instead of managing each device directly.
- TAM: A TAM is responsible for performing lifecycle management activity on TA's on behalf of Service Providers and Device Administrators. This includes creation and deletion of TA's, and may include, for example, over-the-air updates to keep an SP's TAs up-to-date and clean up when a version should be removed. TAMs may provide services that make it easier for SPs or DAs to use the TAM's service to manage multiple devices, although that is not required of a TAM.

The TAM performs its management of TA's through an interaction with a Device's TEEP Broker. As shown in #notionalarch, the TAM cannot directly contact a Device, but must wait for a the TEEP Broker or a Client Application 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 SPs, or a TAM may be private, and accessible by only one or a limited number of SPs.

It is expected that manufacturers and carriers will run their own private TAM. Another example of a private TAM is a TAM running as a Software-as-a-Service (SaaS) within an SP.

A SP 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 an authorized Trust Anchor in the Device. A SP or Device Administrator may run their own TAM, however the Devices they wish to manage must include this TAM's public key in the Trust Anchor list.

A SP or Device Administrator is free to utilize multiple TAMs. This may be required for a SP to manage multiple different types of devices from different manufacturers, or devices on different carriers, since the Trust Anchor list on these different devices may contain different TAMs. A Device Administrator may be able to add their own TAM's public key or certificate to the Trust Anchor list 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 Devices Trust Anchor list. A TAM may set up a relationship with device manufacturers or carriers to have them install the TAM's keys in their device's Trust Anchor list. 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: The TEEP Broker is an application running in a Rich Execution Environment (REE) that enables the message protocol exchange between a TAM and a TEE in a device. The 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.

A Client Application is expected to communicate with a TAM to request TAs that it needs to use. The Client Application needs to pass the messages from the TAM to TEEs in the device. This calls for a component in the REE that Client Applications can use to pass messages to TEEs. The TEEP Broker is thus an application in the REE or software library that can relay messages from a Client Application to a TEE in the device. A device usually comes with only one active TEE. A TEE may provide such a Broker to the device manufacturer to be bundled in devices. Such a TEE must also include a Broker counterpart, namely, a TEEP Agent inside the TEE, to parse TAM messages sent through the Broker. A TEEP Broker is generally acting as a dummy relaying box with just the TEE interacting capability; it doesn't need and shouldn't parse protocol messages.

- **TEEP Agent:** the TEEP Agent is a processing module running inside a TEE that receives TAM requests that are 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 by a TEEP Agent back to a TEEP Broker.
- **Certification Authority (CA):** Certificate-based credentials used for authenticating a device, a TAM and an SP. A device embeds a list of root certificates (Trust Anchors), from trusted CAs that a TAM will be validated against. A TAM will remotely attest a device by checking whether a device comes with a certificate from a CA that the TAM trusts. 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.

5.2. Different Renditions of TEEP Architecture

There is nothing prohibiting a device from implementing multiple TEEs. In addition, some TEEs (for example, SGX) present themselves as separate containers within memory without a controlling manager within the TEE. In these cases, the rich operating system hosts multiple TEEP brokers, where each broker manages a particular TEE or set of TEEs. Enumeration and access to the appropriate broker is up to the rich OS and the applications. Verification that the correct TA has been reached then becomes a matter of properly verifying TA attestations, which are unforgeable. The multiple TEE approach is shown in the diagram below. For brevity, TEEP Broker 2 is shown interacting with only one TAM and UA, but no such limitation is intended to be implied in the architecture.

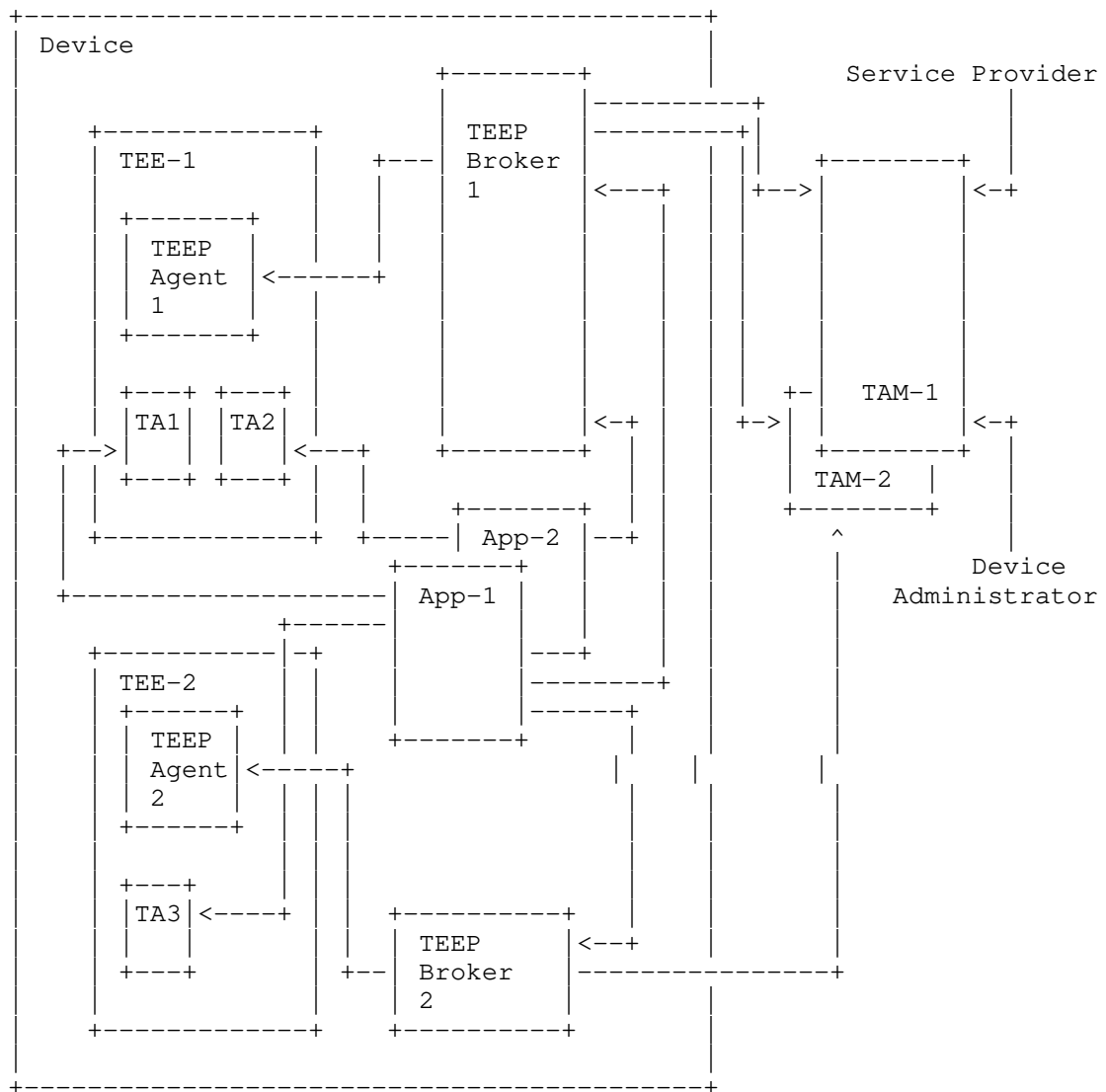


Figure 2: Notional Architecture of TEEP with multiple TEEs

In the diagram above, TEEP Broker 1 controls interactions with the TA's in TEE-1, and TEEP Broker 2 controls interactions with the TA's 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 TEE's may be physically separated, with wholly different resources, there may be no need for TEEP Brokers to share information on installed TAs or

resource usage. However, the architecture guarantees that the TAM will receive all the relevant information from the TEEP Broker to which it communicates.

5.3. Multiple TAMs and Relationship to TAs

As shown in Figure 2, the TEEP Broker provides connections from the TEE and the Client App to one or more TAMs. The selection of which TAM to communicate with is dependent on information from the Client App and is directly related to the TA.

When a SP offers a service which requires a TA, the SP associates that service with a specific TA. The TA itself is digitally signed, protecting its integrity, but the signature also links the TA back to the signer. The signer is usually the SP, but in some cases may be another party that the SP trusts. The SP selects one or more TAMs through which to offer their service, and communicates the information of the service and the specific client apps and TAs to the TAM.

The SP chooses TAMs based upon the markets into which the TAM can provide access. There may be TAMs that provide services to specific types of mobile devices, or mobile device operating systems, or specific geographical regions or network carriers. A SP may be motivated to utilize multiple TAMs for its service in order to maximize market penetration and availability on multiple types of devices. This likely means that the same service will be available through multiple TAMs.

When the SP publishes the Client App to an app store or other app repositories, the SP binds the Client App with a manifest that identifies what TAMs can be contacted for the TA. In some situations, an SP may use only a single TAM - this is likely the case for enterprise applications or SPs serving a closed community. For broad public apps, there will likely be multiple TAMs in the 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 will include different TAMs that support this SP's client app and TA. Multiple TAMs allow the SP to provide thier service and this app (and TA) to multiple different devices.

When the TEEP Broker receives a request to contact the TAM for a Client App in order to install a TA, a list of TAMs may be provided. The TEEP Broker selects a single TAM that is consistent with the list of trusted TAMs (trust anchors) provisioned on the device. For any client app, there should be only a single TAM for the TEEP Broker to contact. This is also the case when a Client App uses multiple TAs,

or when one TA depends on another TA in a software dependency (see section TBD). The reason is that the SP should provide each TAM that it places in the Client App's manifest all the TAs that the app requires. There is no benefit to going to multiple different TAMs, and there is no need for a special TAM to be contacted for a specific TA.

[Note: This should always be the case. When a particular device or TEE supports only a special proprietary attestation mechanism, then a specific TAM will be needed that supports that attestation scheme. The TAM should also support standard attestation signatures as well. It is highly unlikely that a set of TAs would use different proprietary attestation mechanisms since a TEE is likely to support only one such proprietary scheme.]

[Note: This situation gets more complex in situations where a Client App expects another application or a device to already have a specific TA installed. This situation does not occur with SGX, but could occur in situations where the secure world maintains a trusted operating system and runs an entire trusted system with multiple TAs running. This requires more discussion.]

5.4. Client Apps, Trusted Apps, and Personalization Data

In TEEP, there is an explicit relationship and dependence between the client app in the REE and one or more TAs in the TEE, as shown in Figure 2. From the perspective of a device user, a client app that uses one or more TA's in a TEE appears no different from any other untrusted application in the REE. However, the way the client app and its corresponding TA's are packaged, delivered, and installed on the device can vary. The variations depend on whether the client app and TA are bundled together or are provided separately, and this has implications to the management of the TAs in the TEE. In addition to the client app and TA, the TA and/or TEE may require some additional data to personalize the TA to the service provider or the device user. This personalization data is dependent on the TEE, the TA and the SP; an example of personalization data might be username and password of the device user's account with the SP, or a secret symmetric key used to by the TA to communicate with the SP. The personalization data must be encrypted to preserve the confidentiality of potentially sensitive data contained within it. Other than this requirement to support confidentiality, TEEP place no limitations or requirements on the personalization data.

There are three possible cases for bundling of the Client App, TA, and personalization data:

1. The Client App, TA, and personalization data are all bundled together in a single package by the SP and provided to the TEEP Broker through the TAM.
2. The Client App and the TA are bundled together in a single binary, which the TAM or a publicly accessible app store maintains in repository, and the personalization data is separately provided by the SP. In this case, the personalization data is collected by the TAM and included in the InstallTA message to the TEEP Broker.
3. All components are independent. The device user installs the Client App through some independent or device-specific mechanism, and the TAM provides the TA and personalization data from the SP. Delivery of the TA and personalization data may be combined or separate.

5.5. Examples of Application Delivery Mechanisms in Existing TEEs

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

In Intel Software Guard Extensions (SGX), the Client App and TA are typically bound into the same binary (Case 2). The TA is compiled into the Client App binary using SGX tools, and exists in the binary as a shared library (.so or .dll). The Client App loads the TA into an SGX enclave when the client needs the TA. This organization makes it easy to maintain compatibility between the Client App and the TA, since they are updated together. It is entirely possible to create a Client App that loads an external TA into an SGX enclave and use that TA (Case 3). In this case, the Client App 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, for Case 2 and Case 3, the personalization data is normally loaded into the SGX enclave (the TA) after the TA has started. Although Case 1 is possible with SGX, 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 to receive the encrypted binary, decrypt it, separate it into the three different elements, and then install all three. This installation is complex, because the Client App decrypted inside the TEE must be passed out of the TEE to an installer in the REE which would install the Client App; this assumes that the Client App binary includes the TA code also, otherwise there is a significant problem in getting the SGX enclave code (the TA) from the TEE, through the installer and into the Client App in a trusted fashion. 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 Client App, 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 one another.

[NOTE: Need to add an equivalent discussion for an ARM/TZ implementation]

5.6. TEEP Architectural Support for Client App, TA, and Personalization Data Delivery

This section defines TEEP support for the three different cases for delivery of the Client App, TA, and personalization data.

[Note: discussion of format of this single binary, and who/what is responsible for splitting these things apart, and installing the client app into the REE, the TA into the TEE, and the personalization data into the TEE or TA. Obviously the decryption must be done by the TEE but this may not be supported by all TAs.]

5.7. Entity Relations

This architecture leverages asymmetric cryptography to authenticate a device to a TAM. Additionally, a TEE in a device authenticates a TAM and TA signer. The provisioning of Trust Anchors to a device may differ 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 of the TA signers and TAM providers; it may embed a list of default Trust Anchors that the signer of an allowed TA's signer certificate should chain to. A device administrator may choose to accept a subset of the allowed TAs via consent or action of downloading.

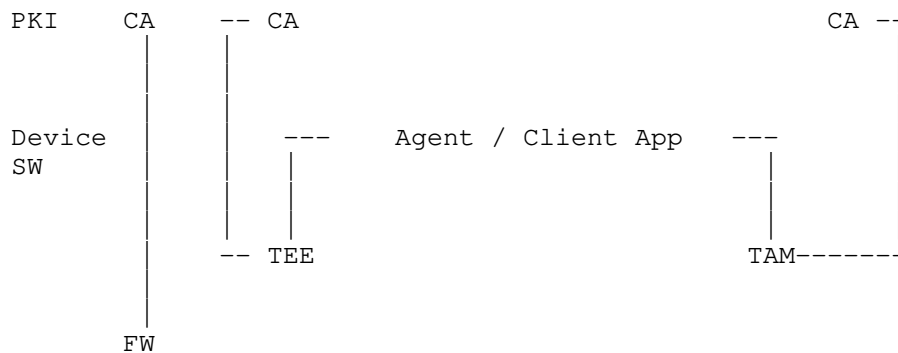


Figure 3: Entities

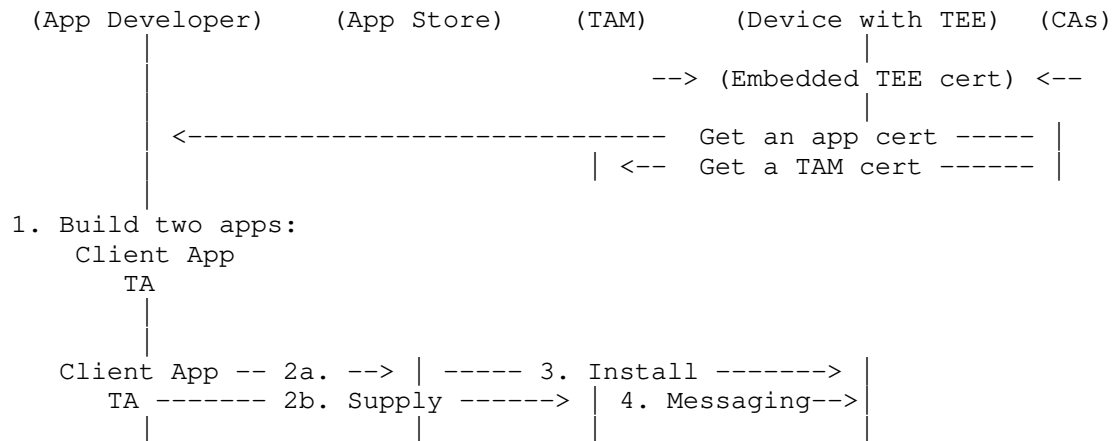


Figure 4: Developer Experience

Figure 4 shows an application developer building two applications: 1) a rich Client Application; 2) a TA that provides some security functions to be run inside a TEE. At step 2, the application developer uploads the Client Application (2a) to an Application Store. The Client Application may optionally bundle the TA binary. Meanwhile, the application developer may provide its TA to a TAM provider that will be managing the TA in various devices. 3. A user will go to an Application Store to download the Client Application. The Client Application will trigger TA installation by initiating communication with a TAM. This is the step 4. The Client Application will get messages from TAM, and interacts with device TEE via an Agent.

The following diagram shows a system diagram about the entity relationships between CAs, TAMs, SPs and devices.

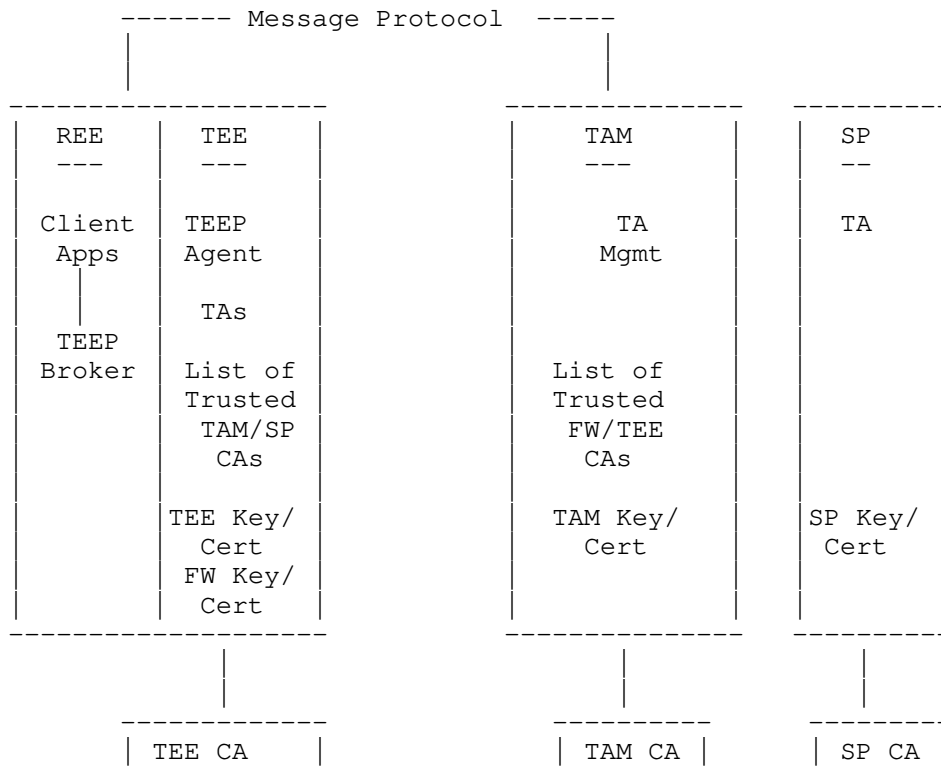


Figure 5: Keys

In the previous diagram, different CAs can be used for different types of certificates. Messages are always signed, where the signer key is the message originator's private key such as that of a TAM, the private key of trusted firmware (TFW), or a TEE's private key.

The main components consist of a set of standard messages created by a TAM to deliver TA 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. The networking functionality must be delegated to a rich Client Application. Client Applications will need to rely on an agent in the REE to interact with a TEE for message exchanges. Consequently, a TAM generally communicates with a Client Application about how it gets messages that originate from a TEE inside a device. Similarly, a TA or TEE generally gets messages from a TAM via some Client Application,

namely, a TEEP Broker in this protocol architecture, not directly from the network.

It is imperative to have an interoperable protocol to communicate with different TAMs and different TEEs in different devices. This is the role of the Broker, which is a software component that bridges communication between a TAM and a TEE. Furthermore the Broker communicates with a Agent inside a TEE that is responsible to process TAM requests. The Broker in REE does not need to know the actual content of messages except for the TEE routing information.

5.8. Trust Anchors in TEE

Each TEE comes with a trust store that contains a whitelist of Trust Anchors that are used to validate a TAM's certificate. A TEE will accept a TAM to create new Security Domains and install new TAs on behalf of an SP only if the TAM's certificate is chained to one of the root CA certificates in the TEE's trust store.

A TEE's trust store is typically preloaded at manufacturing time. It is out of the scope in this document to specify how the trust anchors should be updated when a new root certificate should be added or existing one should be updated or removed. A device manufacturer is expected to provide its TEE trust anchors live update or out-of-band update to Device Administrators.

When trust anchor update is carried out, it is imperative that any update must maintain integrity where only authentic trust anchor list from a device manufacturer or a Device Administrator is accepted. This calls for a complete lifecycle flow in authorizing who can make trust anchor update and whether a given trust anchor list are non-tampered from the original provider. The signing of a trust anchor list for integrity check and update authorization methods are desirable to be developed. This can be addressed outside of this architecture document.

Before a TAM can begin operation in the marketplace to support a device with a particular TEE, it must obtain a TAM certificate from a CA that is listed in the trust store of the TEE.

5.9. Trust Anchors in TAM

The Trust Anchor store in a TAM consists of a list of CA certificates that sign various device TEE certificates. A TAM will accept a device for TA management if the TEE in the device uses a TEE certificate that is chained to a CA that the TAM trusts.

5.10. Keys and Certificate Types

This architecture leverages the following credentials, which allow delivering end-to-end security without relying on any transport security.

Key Entity Name	Location	Issuer	Checked Against	Cardinality
1. TFW key pair and certificate	Device secure storage	FW CA	A whitelist of FW root CA trusted by TAMs	1 per device
2. TEE key pair and certificate	Device TEE	TEE CA under a root CA	A whitelist of TEE root CA trusted by TAMs	1 per device
3. TAM key pair and certificate	TAM provider	TAM CA under a root CA	A whitelist of TAM root CA embedded in TEE	1 or multiple can be used by a TAM
4. SP key pair and certificate	SP	SP signer CA	A SP uses a TAM. TA is signed by a SP signer. TEE delegates trust of TA to TAM. SP signer is associated with a TA as the owner.	1 or multiple can be used by a TAM

Figure 6: Key and Certificate Types

1. TFW key pair and certificate: A key pair and certificate for evidence of trustworthy firmware in a device. This key pair is optional for TEEP architecture. Some TEE may present its trusted attributes to a TAM using signed attestation with a TFW key. For example, a platform that uses a hardware based TEE can have attestation data signed by a hardware protected TFW key.
 - o Location: Device secure storage
 - o Supported Key Type: RSA and ECC
 - o Issuer: OEM CA

- o Checked Against: A whitelist of FW root CA trusted by TAMs
 - o Cardinality: One per device
2. TEE key pair and certificate: It is used for device attestation to a remote TAM and SP.
- o This key pair is burned into the device by the device manufacturer. The key pair and its certificate are valid for the expected lifetime of the device.
 - o Location: Device TEE
 - o Supported Key Type: RSA and ECC
 - o Issuer: A CA that chains to a TEE root CA
 - o Checked Against: A whitelist of TEE root CAs trusted by TAMs
 - o Cardinality: One per device
3. TAM key pair and certificate: A TAM provider acquires a certificate from a CA that a TEE trusts.
- o Location: TAM provider
 - o Supported Key Type: RSA and ECC.
 - o Supported Key Size: RSA 2048-bit, ECC P-256 and P-384. Other sizes should be anticipated in future.
 - o Issuer: TAM CA that chains to a root CA
 - o Checked Against: A whitelist of TAM root CAs embedded in a TEE
 - o Cardinality: One or multiple can be used by a TAM
4. SP key pair and certificate: An SP uses its own key pair and certificate to sign a TA.
- o Location: SP
 - o Supported Key Type: RSA and ECC
 - o Supported Key Size: RSA 2048-bit, ECC P-256 and P-384. Other sizes should be anticipated in future.
 - o Issuer: An SP signer CA that chains to a root CA

- o Checked Against: An SP uses a TAM. A TEE trusts an SP by validating trust against a TAM that the SP uses. A TEE trusts a TAM to ensure that a TA is trustworthy.
- o Cardinality: One or multiple can be used by an SP

5.11. Scalability

This architecture uses a PKI. Trust Anchors exist on the devices to enable the TEE to authenticate TAMs, and TAMs use Trust Anchors to authenticate TEEs. Since 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.

5.12. Message Security

Messages created by a TAM are used to deliver TA management commands to a device, and device attestation and messages created by the device TEE to respond to TAM messages.

These messages are signed end-to-end and are typically encrypted such that only the targeted device TEE or TAM is able to decrypt and view the actual content.

5.13. Security Domain

No security domain (SD) is explicitly assumed in a TEE for TA management. Some TEE, for example, some TEE compliant with Global Platform (GP), may continue to choose to use SD to organize resource partition and security boundaries. It is up to a TEE implementation to decide how a SD is attached to a TA installation, for example, one SD could be created per TA.

5.14. A Sample Device Setup Flow

Step 1: Prepare Images for Devices

1. [TEE vendor] Deliver TEE Image (CODE Binary) to device OEM

2. [CA] Deliver root CA Whitelist

3. [Soc] Deliver TFW Image

Step 2: Inject Key Pairs and Images to Devices

1. [OEM] Generate TFW Key Pair (May be shared among multiple devices)
2. [OEM] Flash signed TFW Image and signed TEE Image onto devices (signed by TFW Key)

Step 3: Set up attestation key pairs in devices

1. [OEM] Flash TFW Public Key and a bootloader key.
2. [TFW/TEE] Generate a unique attestation key pair and get a certificate for the device.

Step 4: Set up Trust Anchors in devices

1. [TFW/TEE] Store the key and certificate encrypted with the bootloader key
2. [TEE vendor or OEM] Store trusted CA certificate list into devices

6. TEEP Broker

A TEE and TAs do not generally have the capability to communicate to the 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 the network communication. Each Client Application in the REE might carry this communication functionality but such functionality must also interact with the TEE for the message exchange.

The TEE interaction will vary according to different TEEs. In order for a Client Application to transparently support different TEEs, it is imperative to have a common interface for a Client Application to invoke for exchanging messages with TEEs.

A shared module in REE comes to meet this need. A TEEP broker is an application running in the REE of the device or an SDK that facilitates communication between a TAM and a TEE. It also provides interfaces for Client Applications to query and trigger TA installation that the application needs to use.

It isn't always that a Client Application directly calls such a Broker to interact with a TEE. A REE Application Installer might carry out TEE and TAM interaction to install all required TAs that a Client Application depends. A Client Application may have a metadata file that describes the TAs it depends on and the associated TAM that each TA installation goes to use. The REE Application Installer can inspect the application metadata file and installs TAs on behalf of the Client Application without requiring the Client Application to run first.

This interface for Client Applications or Application Installers may be commonly in a form of an OS service call for an REE OS. A Client Application or an Application Installer interacts with the device TEE and the TAMs.

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 TA installation.

The Broker doesn't need to parse a message content received from a TAM that should be processed by a TEE. When a device has more than one TEE, one TEEP Broker per TEE could be present in REE. A TEEP Broker interacts with a TEEP Agent inside a TEE.

A TAM message may indicate the target TEE where a TA should be installed. A compliant TEEP protocol should include a target TEE identifier for a TEEP Broker when multiple TEEs are present.

The Broker relays the response messages generated from a TEEP Agent in a TEE to the TAM. The Broker is not expected to handle any network connection with an application or TAM.

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

6.2. TEEP Broker Implementation Consideration

A Provider should consider methods of distribution, scope and concurrency on devices and runtime options when implementing a TEEP Broker. Several non-exhaustive options are discussed below. Providers are encouraged to take advantage of the latest communication and platform capabilities to offer the best user experience.

6.2.1. TEEP Broker Distribution

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

6.2.2. Number of TEEP Brokers

There should be generally only one shared TEEP Broker in a device. The device's TEE vendor will most probably supply one Broker. When multiple TEEs are present in a device, one TEEP Broker per TEE may be used.

When only one Broker is used per device, the Broker provider is responsible to allow multiple TAMs and TEE providers to achieve interoperability. With a standard Broker interface, each TAM can implement its own SDK for its SP Client Applications to work with this Broker.

Multiple independent Broker providers can be used as long as they have standard interface to a Client Application or TAM SDK. Only one Broker is generally expected in a device.

7. Attestation

Attestation is the process through which one entity (an attester) presents a series of claims to another entity (a verifier), and provides sufficient proof that the claims are true. Different verifiers may have different standards for attestation proofs and not all attestations are acceptable to every verifier. TEEP attestations are based upon the use of an asymmetric key pair under the control of the TEE to create digital signatures across a well-defined claim set.

In TEEP, the primary purpose of an attestation is to allow a device to prove to TAMs and SPs that a TEE in the device has particular properties, was built by a particular manufacturer, or is executing a particular TA. Other claims are possible; this architecture specification does not limit the attestation claims, but defines a minimal set of claims required for TEEP to operate properly. Extensions to these claims are possible, but are not defined in the TEEP specifications. Other standards or groups may define the format and semantics of extended claims. The TEEP specification defines the claims format such that these extended claims may be easily included in a TEEP attestation message.

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 algorithms and mechanisms. In order for TEEP to be inclusive, the attestation

format shall allow for both proprietary attestation signatures, as well as a standardized form of attestation signature. Either form of attestation signature may be applied to a set of TEEP claims, and both forms of attestation shall be considered conformant with TEEP. However, it should be recognized that not all TAMs or SPs may be able to process all proprietary forms of attestations. All TAMs and SPs MUST be able to process the TEEP standard attestation format and attached signature.

The attestation formats and mechanisms described and mandated by TEEP shall convey a particular set of cryptographic properties based on minimal assumptions. The cryptographic properties are conveyed by the attestation; however the assumptions are not conveyed within the attestation itself.

The assumptions which may apply to an attestation have to do with the quality of the attestation and the quality and security provided by the TEE, the device, the manufacturer, or others involved in the device or TEE ecosystem. Some of the assumptions that might apply to an attestations include (this may not be a comprehensive list):

- Assumptions regarding the security measures a manufacturer takes when provisioning keys into devices/TEEs;
- Assumptions regarding what hardware and software components have access to the Attestation keys of the TEE;
- Assumptions related to the source or local verification of claims within an attestation prior to a TEE signing a set of claims;
- Assumptions regarding the level of protection afforded to attestation keys against exfiltration, modification, and side channel attacks;
- Assumptions regarding the limitations of use applied to TEE Attestation keys;
- Assumptions regarding the processes in place to discover or detect TEE breeches; and
- Assumptions regarding the revocation and recovery process of TEE attestation keys.

TAMs and SPs must be comfortable with the assumptions that are inherently part of any attestation they accept. Alternatively, any TAM or SP may choose not to accept an attestation generated from a particular manufacturer or device's TEE based on the inherent

assumptions. The choice and policy decisions are left up to the particular TAM/SP.

Some TAMs or SPs may require additional claims in order to properly authorize a device or TEE. These additional claims may help clear up any assumptions for which the TAM/SP wants to alleviate. The specific format for these additional claims are outside the scope of this specification, but the OTrP protocol SHALL allow these additional claims to be included in the attestation messages.

The following sub-sections define the cryptographic properties conveyed by the TEEP attestation, the basic set of TEEP claims required in a TEEP attestation, the TEEP attestation flow between the TAM the device TEE, and some implementation examples of how an attestation key may be realized in a real TEEP device.

7.1. Attestation Cryptographic Properties

The attestation constructed by TEEP must convey certain cryptographic properties from the attester to the verifier; in the case of TEEP, the attestation must convey properties from the device to the TAM and/or SP. The properties required by TEEP include:

- Non-repudiation, Unique Proof of Source - the cryptographic digital signature across the attestation, and optionally along with information in the attestation itself SHALL uniquely identify a specific TEE in a specific device.
- Integrity of claims - the cryptographic digital signature across the attestation SHALL cover the entire attestation including all meta data and all the claims in the attestation, ensuring that the attestation has not be modified since the TEE signed the attestation.

Standard public key algorithms such as RSA and ECDSA digital signatures convey these properties. Group public key algorithms such as EPID can also convey these properties, if the attestation includes a unique device identifier and an identifier for the TEE. Other cryptographic operations used in other attestation schemes may also convey these properties.

The TEEP standard attestation format SHALL use one of the following digital signature formats:

- RSA-2048 with SHA-256 or SHA-384 in RSASSA-PKCS1-v1_5 or PSS format

- RSA-3072 with SHA-256 or SHA-384 in RSASSA-PKCS1-v1_5 or PSS format
- ECDSA-256 using NIST P256 curve using SHA-256
- ECDSA-384 using NIST P384 curve using SHA-384
- HashEdDSA using Ed25519 with SHA-512 (Ed25519ph in RFC8032) and context="TEEP Attestation"
- EdDSA using Ed448 with SHAK256 (Ed448ph in RFC8032) and context="TEEP Attestation"

All TAMs and SPs MUST be able to accept attestations using these algorithms, contingent on their acceptance of the assumptions implied by the attestations.

7.2. TEEP Attestation Structure

For a TEEP attestation to be useful, it must contain an information set allowing the TAM and/or SP to assess the attestation and make a related security policy decision. The structure of the TEEP attestation is shown in the diagram below.

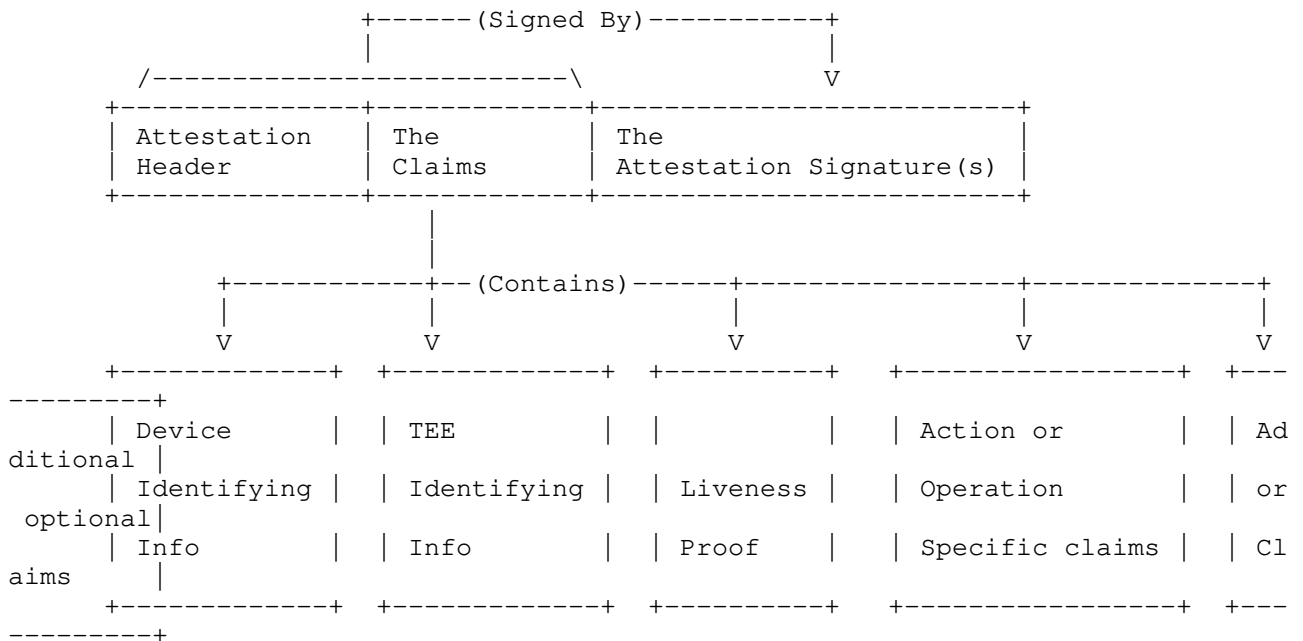


Figure 7: Structure of TEEP Attestation

The Attestation Header SHALL identify the "Attestation Type" and the "Attestation Signature Type" along with an "Attestation Format Version Number." The "Attestation Type" identifies the minimal set of claims that MUST be included in the attestation; this is an

identifier for a profile that defines the claims that should be included in the attestation as part of the "Action or Operation Specific Claims." The "Attestation Signature Type" identifies the type of attestation signature that is attached. The type of attestation signature SHALL be one of the standard signatures types identified by an IANA number, a proprietary signature type identified by an IANA number, or the generic "Proprietary Signature" with an accompanying proprietary identifier. Not all TAMs may be able to process proprietary signatures.

The claims in the attestation are set of mandatory and optional claims. The claims themselves SHALL be defined in an attestation claims dictionary. See the next section on TEEP Attestation Claims. Claims are grouped in profiles under an identifier (Attestation Type), however all attestations require a minimal set of claims which includes:

- Device Identifying Info: TEEP attestations must uniquely identify a device to the TAM and SP. This identifier allows the TAM/SP to provide services unique 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. Additionally, device manufacturer information must be provided to provide better universal uniqueness qualities without requiring globally unique identifiers for all devices.
- TEE Identifying info: The type of TEE that generated this attestation must be identified. Standard TEE types are identified by an IANA number, but also must include version identification information such as the hardware, firmware, and software version of the TEE, as applicable by the TEE type. Structure to the version number is required. TEE manufacturer information for the TEE is required in order to disambiguate the same TEE type created by different manufacturers and resolve potential assumptions around manufacturer provisioning, keying and support for the TEE.
- Liveness Proof: a claim that includes liveness information SHALL be included which may be a large nonce or may be a timestamp and short nonce.
- Action Specific Claims: Certain attestation types shall include specific claims. For example an attestation from a specific TA shall include a measurement, version and signing public key for the TA.
- Additional Claims: (Optional - May be empty set) A TAM or SP may require specific additional claims in order to address potential assumptions, such as the requirement that a device's REE performed

a secure boot, or that the device is not currently in a debug or non-productions state. A TAM may require a device to provide a device health attestation that may include some claims or measurements about the REE. These claims are TAM specific.

7.3. TEEP Attestation Claims

TEEP requires a set of attestation claims that provide sufficient evidence to the TAM and/or SP that the device and its TEE meet certain minimal requirements. Because attestation formats are not yet broadly standardized across the industry, standardization work is currently ongoing, and it is expected that extensions to the attestation claims will be required as new TEEs and devices are created, the set of attestation claims required by TEEP SHALL be defined in an IANA registry. That registry SHALL be defined in the OTrP protocol with sufficient elements to address basic TEEP claims, expected new standard claims (for example from <https://www.ietf.org/id/draft-mandyam-eat-01.txt>), and proprietary claim sets.

7.4. TEEP Attestation Flow

Attestations are required in TEEP under the following flows:

- When a TEE responds with device state information (dsi) to the TAM or SP, including a "GetDeviceState" response, "InstallTA" response, etc.
- When a new key pair is generated for a TA-to-TAM or TA-to-SP communication, the keypair must be covered by an attestation, including "CreateSecurityDomain" response, "UpdateSecurityDomain" response, etc.

7.5. Attestation Key Example

The attestation hierarchy and seed required for TAM protocol operation must be built into the device at manufacture. Additional TEEs can be added post-manufacture using the scheme proposed, but it is outside of the current scope of this document to detail that.

It should be noted that the attestation scheme described is based on signatures. The only decryption that may take place is through the use of a bootloader key.

A boot module generated attestation can be optional where the starting point of device attestation can be at TEE certificates. A TAM can define its policies on what kinds of TEE it trusts if TFW attestation is not included during the TEE attestation.

7.5.1. Attestation Hierarchy Establishment: Manufacture

During manufacture the following steps are required:

1. A device-specific TFW key pair and certificate are burnt into the device. This key pair will be used for signing operations performed by the boot module.
2. TEE images are loaded and include a TEE instance-specific key pair and certificate. The key pair and certificate are included in the image and covered by the code signing hash.
3. The process for TEE images is repeated for any subordinate TEEs, which are additional TEEs after the root TEE that some devices have.

7.5.2. Attestation Hierarchy Establishment: Device Boot

During device boot the following steps are required:

1. The boot module releases the TFW private key by decrypting it with the bootloader key.
2. The boot module verifies the code-signing signature of the active TEE and places its TEE public key into a signing buffer, along with its identifier for later access. For a TEE non-compliant to this architecture, the boot module leaves the TEE public key field blank.
3. The boot module signs the signing buffer with the TFW private key.
4. Each active TEE performs the same operation as the boot module, building up their own signed buffer containing subordinate TEE information.

7.5.3. Attestation Hierarchy Establishment: TAM

Before a TAM can begin operation in the marketplace, it must obtain a TAM certificate from a CA that is registered in the trust store of devices. In this way, the TEE can check the intermediate and root CA and verify that it trusts this TAM to perform operations on the TEE.

8. Algorithm and Attestation Agility

RFC 7696 [RFC7696] outlines the requirements to migrate from one mandatory-to-implement algorithm suite to another over time. This feature is also known as crypto agility. Protocol evolution is

greatly simplified when crypto agility is already considered during the design of the protocol. In the case of the Open Trust Protocol (OTrP) the diverse range of use cases, from trusted app updates for smart phones 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 the OTrP concerns the use of symmetric as well as asymmetric algorithms. Symmetric algorithms are used for encryption of content whereas the asymmetric algorithms are mostly used for signing messages.

In addition to the use of cryptographic algorithms in OTrP 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. TA Trust Check at TEE

A TA binary is signed by a TA signer certificate. This TA signing certificate/private key belongs to the SP, and may be self-signed (i.e., it need not participate in a trust hierarchy). It is the responsibility of the TAM to only allow verified TAs from trusted SPs into the system. Delivery of that TA to the TEE is then the responsibility of the TEE, using the security mechanisms provided by the protocol.

We allow a way for an (untrusted) application to check the trustworthiness of a TA. A TEEP Broker has a function to allow an application to query the information about a TA.

An application in the Rich O/S may perform verification of the TA by verifying the signature of the TA. The GetTAInformation function is available to return the TEE supplied TA signer and TAM signer information to the application. An application can do additional trust checks on the certificate returned for this TA. It might trust the TAM, or require additional SP signer trust chaining.

9.2. One TA Multiple SP Case

A TA for multiple SPs must have a different identifier per SP. They should appear as different TAs when they are installed in the same device.

9.3. Broker Trust Model

A TEEP Broker could be malware in the vulnerable REE. A Client Application will connect its TAM provider for required TA installation. It gets command messages from the TAM, and passes the message to the Broker.

The architecture enables the TAM to communicate with the device's TEE to manage TAs. All TAM messages are signed and sensitive data is encrypted such that the TEEP Broker cannot modify or capture sensitive data.

9.4. Data Protection at TAM and TEE

The TEE implementation provides protection of data on the device. It is the responsibility of the TAM to protect data on its servers.

9.5. Compromised CA

A root CA for TAM certificates might get compromised. Some TEE trust anchor update mechanism is expected from device OEMs. A compromised intermediate CA is covered by OCSP stapling and OCSP validation check in the protocol. A TEE should validate certificate revocation about a TAM certificate chain.

If the root CA of some TEE device certificates is compromised, these devices might be rejected by a TAM, which is a decision of the TAM implementation and policy choice. Any intermediate CA for TEE device certificates SHOULD be validated by TAM with a Certificate Revocation List (CRL) or Online Certificate Status Protocol (OCSP) method.

9.6. Compromised TAM

The TEE SHOULD use validation of the supplied TAM certificates and OCSP stapled data to validate that the TAM is trustworthy.

Since PKI is used, the integrity of the clock within the TEE determines the ability of the TEE to reject an expired TAM certificate, or revoked TAM certificate. Since OCSP stapling includes signature generation time, certificate validity dates are compared to the current time.

9.7. Certificate Renewal

TFW and 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 update. On the other hand, it is imperative that OEMs or device providers plan for support of Trust Anchor update in their shipped devices.

10. IANA Considerations

This document does not require actions by IANA.

11. Acknowledgements

The authors thank Dave Thaler for his very thorough review and many important suggestions. Most content of this document is split from a previously combined OTrP protocol document [I-D.ietf-teep-opentrustprotocol]. We thank the former co-authors Nick Cook and Minho Yoo for the initial document content, and contributors Brian Witten, Tyler Kim, and Alin Mutu.

12. References

12.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

12.2. Informative References

- [GPTEE] Global Platform, "GlobalPlatform Device Technology: TEE System Architecture, v1.1", Global Platform GPD_SPE_009, January 2017, <<https://globalplatform.org/specs-library/tee-system-architecture-v1-1/>>.
- [I-D.ietf-teep-opentrustprotocol] Pei, M., Atyeo, A., Cook, N., Yoo, M., and H. Tschofenig, "The Open Trust Protocol (OTrP)", draft-ietf-teep-opentrustprotocol-03 (work in progress), May 2019.
- [RFC6024] Reddy, R. and C. Wallace, "Trust Anchor Management Requirements", RFC 6024, DOI 10.17487/RFC6024, October 2010, <<https://www.rfc-editor.org/info/rfc6024>>.

[RFC7696] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, DOI 10.17487/RFC7696, November 2015, <<https://www.rfc-editor.org/info/rfc7696>>.

Appendix A. History

RFC EDITOR: PLEASE REMOVE THIS SECTION

IETF Drafts

draft-00: - Initial working group document

Authors' Addresses

Mingliang Pei
Symantec

EMail: mingliang_pei@symantec.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

David Wheeler
Intel

EMail: david.m.wheeler@intel.com

Andrew Atyeo
Intercede

EMail: andrew.atyeo@intercede.com

Liu Dapeng
Alibaba Group

EMail: maxpassion@gmail.com

TEEP
Internet-Draft
Intended status: Informational
Expires: 21 October 2022

M. Pei
Broadcom
H. Tschofenig
Arm Limited
D. Thaler
Microsoft
D. Wheeler
Amazon
19 April 2022

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 SGX	16
4.4.2. Example: Application Delivery Mechanisms in Arm TrustZone	17
4.5. Entity Relations	17
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

10. IANA Considerations	34
11. Contributors	34
12. Acknowledgements	34
13. Informative References	34
Authors' Addresses	36

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 pwner) 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.

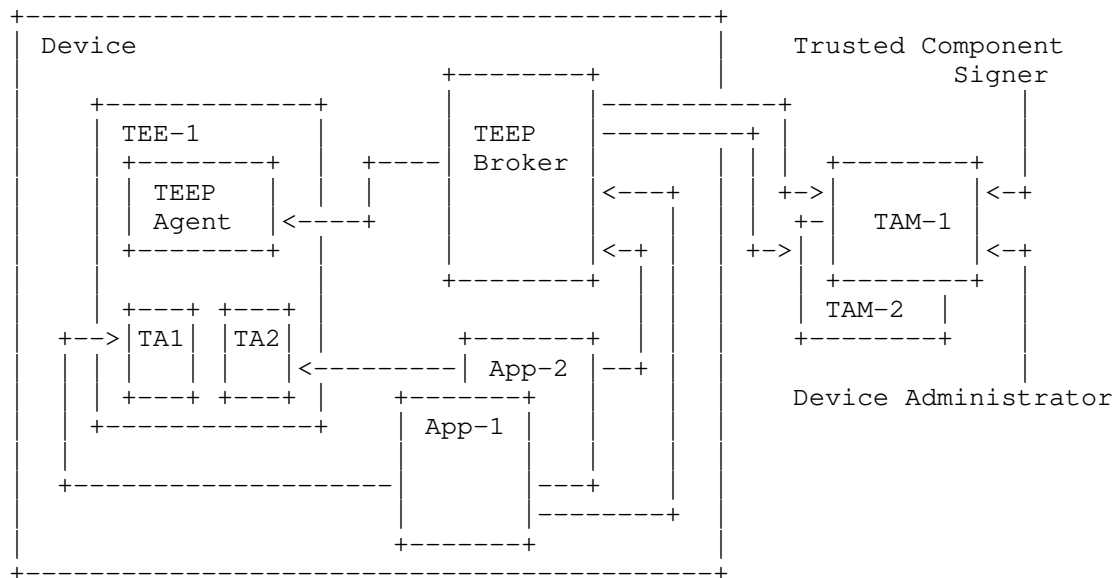


Figure 1: Notional Architecture of TEEP

- * 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.

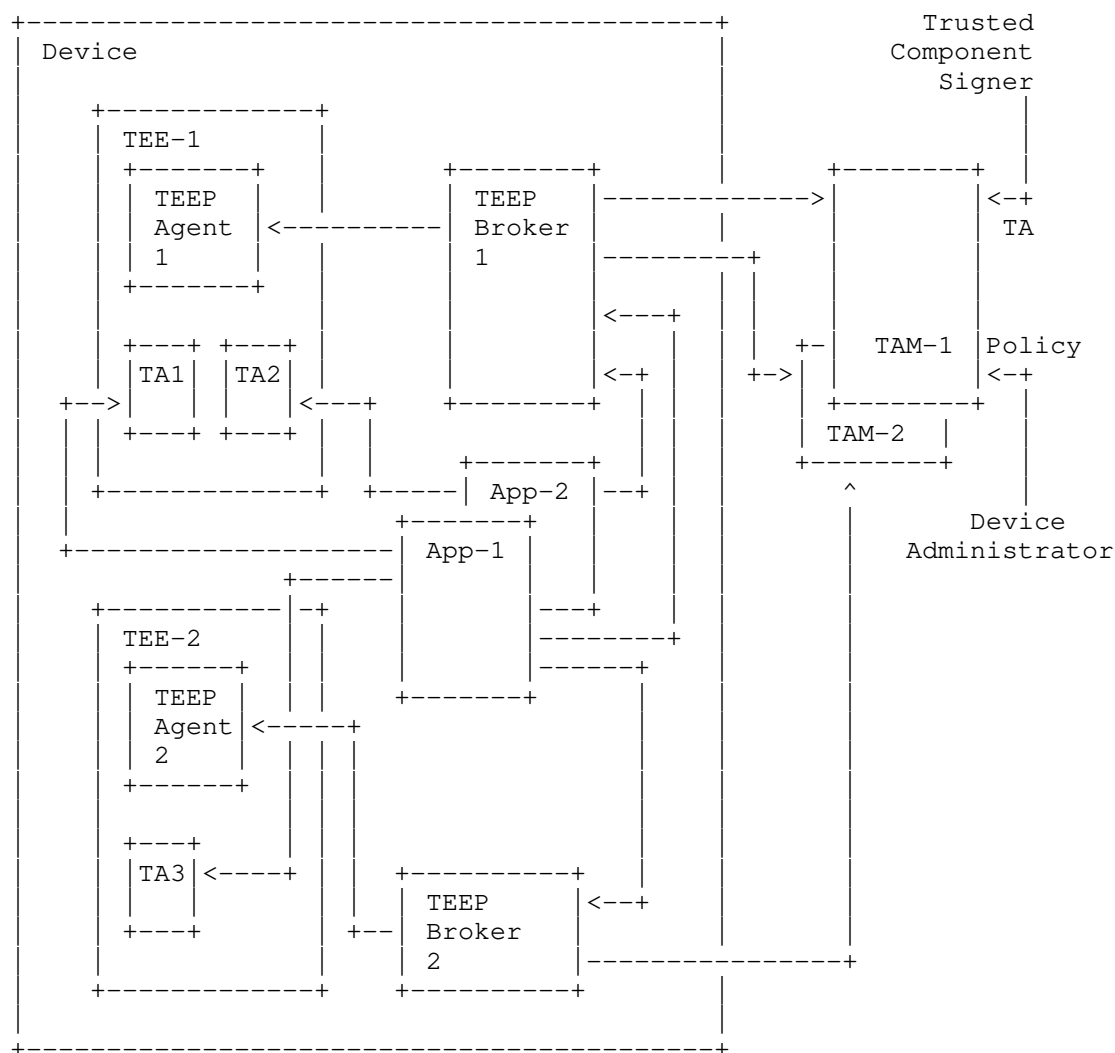


Figure 2: Notional Architecture of TEEP with multiple TEEs

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.

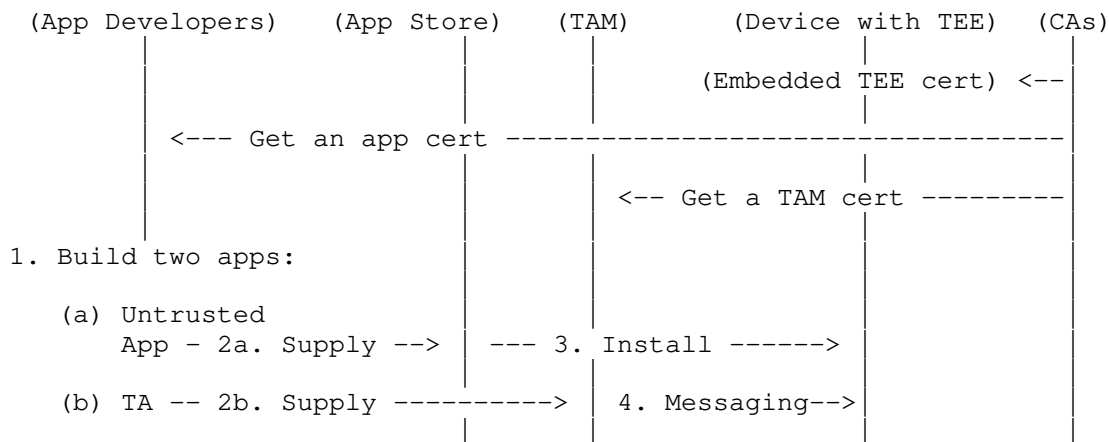


Figure 3: Example Developer Experience

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.

Purpose	Cardinality & Location of Private Key	Private Key Signs	Location of Trust Anchor Store
Authenticating TEEP Agent	1 per TEE	TEEP responses	TAM
Authenticating TAM	1 per TAM	TEEP requests	TEEP Agent
Code Signing	1 per Trusted Component Signer	TA binary	TEE

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.

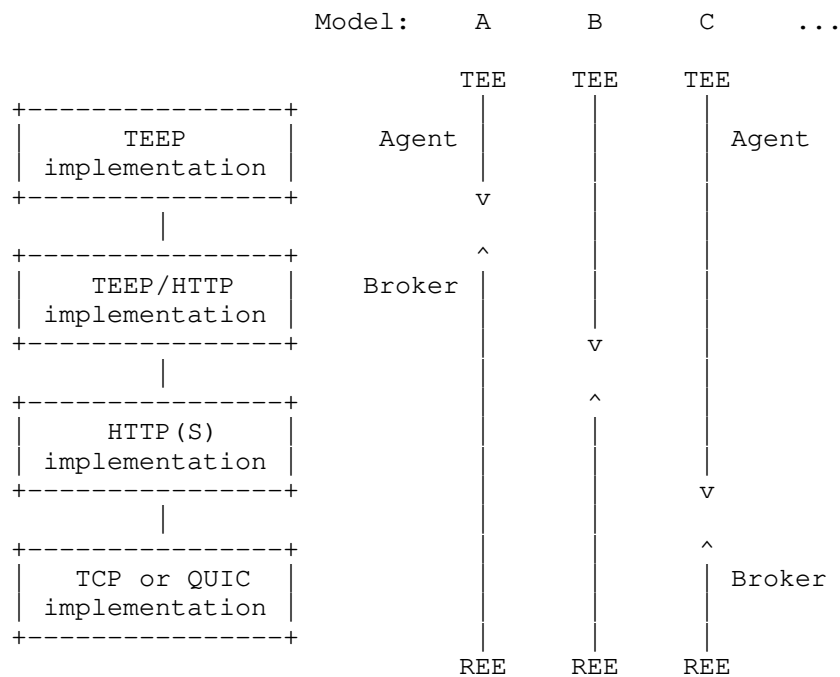


Figure 5: TEEP Broker Models

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.

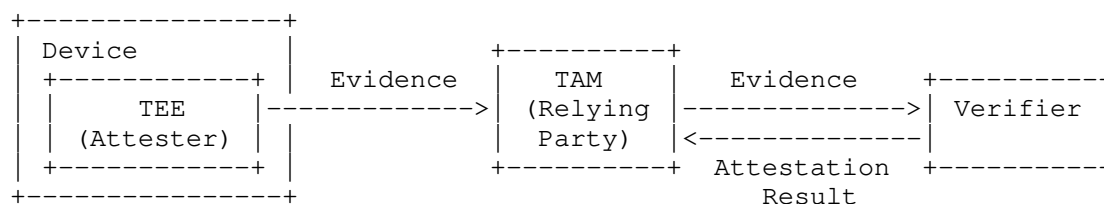


Figure 6: TEEP Attestation Roles

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 smart phones 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, Juerген Schoenwaelder, Nicolae Paladi, Sorin Faibish, Ned Smith, Russ Housley, Jeremy O'Donoghue, and Anders Rundgren for their feedback.

13. Informative References

[CC-Overview]

Confidential Computing Consortium, "Confidential Computing: Hardware-Based Trusted Execution for Applications and Data", January 2021, <https://confidentialcomputing.io/wp-content/uploads/sites/85/2021/03/confidentialcomputing_outreach_whitepaper-8-5x11-1.pdf>.

[CC-Technical-Analysis]

Confidential Computing Consortium, "A Technical Analysis of Confidential Computing, v1.2", October 2021, <<https://confidentialcomputing.io/wp-content/uploads/sites/85/2022/01/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.2.pdf>>.

[GPTEE]

GlobalPlatform, "GlobalPlatform Device Technology: TEE System Architecture, v1.1", GlobalPlatform GPD_SPE_009, January 2017, <<https://globalplatform.org/specs-library/tee-system-architecture-v1-1/>>.

[GSMA]

GSM Association, "GP.22 RSP Technical Specification, Version 2.2.2", June 2020, <<https://www.gsma.com/esim/wp-content/uploads/2020/06/SGP.22-v2.2.2.pdf>>.

[I-D.ietf-rats-architecture]

Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote Attestation Procedures Architecture", Work in Progress, Internet-Draft, draft-ietf-rats-architecture-15, 8 February 2022, <<https://www.ietf.org/archive/id/draft-ietf-rats-architecture-15.txt>>.

[I-D.ietf-suit-architecture]

Moran, B., Tschofenig, H., Brown, D., and M. Meriac, "A Firmware Update Architecture for Internet of Things", Work in Progress, Internet-Draft, draft-ietf-suit-architecture-16, 27 January 2021, <<https://www.ietf.org/archive/id/draft-ietf-suit-architecture-16.txt>>.

[I-D.ietf-suit-manifest]

Moran, B., Tschofenig, H., Birkholz, H., and K. Zandberg, "A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest", Work in Progress, Internet-Draft, draft-ietf-suit-manifest-16, 25 October 2021, <<https://www.ietf.org/archive/id/draft-ietf-suit-manifest-16.txt>>.

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

Thaler, D., "HTTP Transport for Trusted Execution Environment Provisioning: Agent Initiated Communication", Work in Progress, Internet-Draft, draft-ietf-teep-otrp-over-http-13, 28 February 2022, <<https://www.ietf.org/archive/id/draft-ietf-teep-otrp-over-http-13.txt>>.

[I-D.ietf-teep-protocol]

Tschofenig, H., Pei, M., Wheeler, D., Thaler, D., and A. Tsukamoto, "Trusted Execution Environment Provisioning (TEEP) Protocol", Work in Progress, Internet-Draft, draft-ietf-teep-protocol-08, 7 March 2022, <<https://www.ietf.org/archive/id/draft-ietf-teep-protocol-08.txt>>.

[OTRP]

GlobalPlatform, "Open Trust Protocol (OTrP) Profile v1.1", GlobalPlatform GPD_SPE_123, July 2020, <<https://globalplatform.org/specs-library/tee-management-framework-open-trust-protocol/>>.

[RFC4949]

Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <<https://www.rfc-editor.org/info/rfc4949>>.

- [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 (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC6024] Reddy, R. and C. Wallace, "Trust Anchor Management Requirements", RFC 6024, DOI 10.17487/RFC6024, October 2010, <<https://www.rfc-editor.org/info/rfc6024>>.
- [RFC7696] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, DOI 10.17487/RFC7696, November 2015, <<https://www.rfc-editor.org/info/rfc7696>>.
- [SGX] Intel, "Intel(R) Software Guard Extensions (Intel (R) SGX)", n.d., <<https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html>>.
- [TrustZone] Arm, "Arm TrustZone Technology", n.d., <<https://developer.arm.com/ip-products/security-ip/trustzone>>.

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

TEEP
Internet-Draft
Intended status: Informational
Expires: November 16, 2019

M. Pei
Symantec
A. Atyeo
Intercede
N. Cook
ARM Ltd.
M. Yoo
IoTrust
H. Tschofenig
ARM Ltd.
May 15, 2019

The Open Trust Protocol (OTrP)
draft-ietf-teep-opentrustprotocol-03.txt

Abstract

This document specifies the Open Trust Protocol (OTrP), a protocol that follows the Trust Execution Environment Provisioning (TEEP) architecture and provides a message protocol that provisions and manages Trusted Applications into a device with a Trusted Execution Environment (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 November 16, 2019.

Copyright Notice

Copyright (c) 2019 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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	5
2. Requirements Language	6
3. Terminology	6
3.1. Definitions	6
3.2. Abbreviations	6
4. OTrP Entities and Trust Model	6
4.1. System Components	6
4.2. Trust Anchors in TEE	7
4.3. Trust Anchors in TAM	7
4.4. Keys and Certificate Types	7
5. Protocol Scope and Entity Relations	10
5.1. A Sample Device Setup Flow	12
5.2. Derived Keys in The Protocol	12
5.3. Security Domain Hierarchy and Ownership	13
5.4. SD Owner Identification and TAM Certificate Requirements	13
5.5. Service Provider Container	14
6. OTrP Broker	15
6.1. Role of OTrP Broker	15
6.2. OTrP Broker and Global Platform TEE Client API	16
6.3. OTrP Broker Implementation Consideration	16
6.3.1. OTrP Broker Distribution	16
6.3.2. Number of OTrP Broker	16
6.4. OTrP Broker Interfaces for Client Applications	17
6.4.1. ProcessOTrPMessage call	17
6.4.2. GetTAInformation call	17
6.5. Sample End-to-End Client Application Flow	20
6.5.1. Case 1: A New Client Application Uses a TA	20
6.5.2. Case 2: A Previously Installed Client Application Calls a TA	21
7. OTrP Messages	22
7.1. Message Format	22
7.2. Message Naming Convention	22
7.3. Request and Response Message Template	23
7.4. Signed Request and Response Message Structure	23
7.4.1. Identifying Signing and Encryption Keys for JWS/JWE Messaging	25
7.5. JSON Signing and Encryption Algorithms	25
7.5.1. Supported JSON Signing Algorithms	27

7.5.2.	Support JSON Encryption Algorithms	27
7.5.3.	Supported JSON Key Management Algorithms	27
7.6.	Common Errors	28
7.7.	OTrP Message List	28
7.8.	OTrP Request Message Routing Rules	29
7.8.1.	SP Anonymous Attestation Key (SP AIK)	29
8.	Transport Protocol Support	29
9.	Detailed Messages Specification	30
9.1.	GetDeviceState	30
9.1.1.	GetDeviceStateRequest message	30
9.1.2.	Request processing requirements at a TEE	31
9.1.3.	Firmware Signed Data	32
9.1.3.1.	Supported Firmware Signature Methods	33
9.1.4.	Post Conditions	33
9.1.5.	GetDeviceStateResponse Message	33
9.1.6.	Error Conditions	38
9.1.7.	TAM Processing Requirements	39
9.2.	Security Domain Management	40
9.2.1.	CreateSD	40
9.2.1.1.	CreateSDRequest Message	40
9.2.1.2.	Request Processing Requirements at a TEE	43
9.2.1.3.	CreateSDResponse Message	44
9.2.1.4.	Error Conditions	45
9.2.2.	UpdateSD	46
9.2.2.1.	UpdateSDRequest Message	46
9.2.2.2.	Request Processing Requirements at a TEE	49
9.2.2.3.	UpdateSDResponse Message	51
9.2.2.4.	Error Conditions	52
9.2.3.	DeleteSD	52
9.2.3.1.	DeleteSDRequest Message	53
9.2.3.2.	Request Processing Requirements at a TEE	55
9.2.3.3.	DeleteSDResponse Message	56
9.2.3.4.	Error Conditions	57
9.3.	Trusted Application Management	57
9.3.1.	InstallTA	58
9.3.1.1.	InstallTARequest Message	59
9.3.1.2.	InstallTAResponse Message	61
9.3.1.3.	Error Conditions	62
9.3.2.	UpdateTA	63
9.3.2.1.	UpdateTARequest Message	64
9.3.2.2.	UpdateTAResponse Message	66
9.3.2.3.	Error Conditions	67
9.3.3.	DeleteTA	68
9.3.3.1.	DeleteTARequest Message	68
9.3.3.2.	Request Processing Requirements at a TEE	70
9.3.3.3.	DeleteTAResponse Message	70
9.3.3.4.	Error Conditions	71
10.	Response Messages a TAM May Expect	72

11. Basic Protocol Profile	73
12. Attestation Implementation Consideration	73
12.1. OTrP Trusted Firmware	74
12.1.1. Attestation signer	74
12.1.2. TFW Initial Requirements	74
12.2. TEE Loading	74
12.3. Attestation Hierarchy	75
12.3.1. Attestation Hierarchy Establishment: Manufacture . .	75
12.3.2. Attestation Hierarchy Establishment: Device Boot . .	75
12.3.3. Attestation Hierarchy Establishment: TAM	76
13. IANA Considerations	76
13.1. Error Code List	77
13.1.1. TEE Signed Error Code List	77
14. Security Consideration	78
14.1. Cryptographic Strength	78
14.2. Message Security	79
14.3. TEE Attestation	79
14.4. TA Protection	79
14.5. TA Personalization Data	80
14.6. TA Trust Check at TEE	80
14.7. One TA Multiple SP Case	81
14.8. OTrP Broker Trust Model	81
14.9. OCSP Stapling Data for TAM Signed Messages	81
14.10. Data Protection at TAM and TEE	81
14.11. Privacy Consideration	82
14.12. Threat Mitigation	82
14.13. Compromised CA	83
14.14. Compromised TAM	83
14.15. Certificate Renewal	83
15. Acknowledgements	83
16. References	84
16.1. Normative References	84
16.2. Informative References	84
Appendix A. Sample Messages	85
A.1. Sample Security Domain Management Messages	85
A.1.1. Sample GetDeviceState	85
A.1.1.1. Sample GetDeviceStateRequest	85
A.1.1.2. Sample GetDeviceStateResponse	85
A.1.2. Sample CreateSD	89
A.1.2.1. Sample CreateSDRequest	89
A.1.2.2. Sample CreateSDResponse	92
A.1.3. Sample UpdateSD	93
A.1.3.1. Sample UpdateSDRequest	94
A.1.3.2. Sample UpdateSDResponse	95
A.1.4. Sample DeleteSD	95
A.1.4.1. Sample DeleteSDRequest	95
A.1.4.2. Sample DeleteSDResponse	97
A.2. Sample TA Management Messages	99

A.2.1. Sample InstallTA	99
A.2.1.1. Sample InstallTAResponse	99
A.2.1.2. Sample InstallTAResponse	100
A.2.2. Sample UpdateTA	102
A.2.2.1. Sample UpdateTAResponse	102
A.2.2.2. Sample UpdateTAResponse	103
A.2.3. Sample DeleteTA	106
A.2.3.1. Sample DeleteTAResponse	106
A.2.3.2. Sample DeleteTAResponse	108
A.3. Example OTrP Broker Option	110
Appendix B. Contributors	110
Authors' Addresses	110

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 an TEE ecosystem, different device vendors may use different TEE implementations. Different application providers or device administrators may choose to use different TAM providers. There calls for an interoperable protocol for managing TAs running in different TEEs of various devices is needed.

The Trusted Execution Environment Provisioning (TEEP) architecture document [TEEPArch] has set to provide a design guidance for such an interoperable protocol. This document specifies an Open Trust Protocol (OTrP) that follows the architecture guidance.

OTrP defines a mutual trust message protocol between a TAM and a TEE and relies on IETF-defined end-to-end security mechanisms, namely JSON Web Encryption (JWE), JSON Web Signature (JWS), and JSON Web Key (JWK). Other message encoding methods may be supported.

This specification defines message payloads exchanged between devices and a TAM. The messages are designed in anticipation of the use of the most common transport methods such as HTTPS.

Each TA binary and configuration data can be from either of two sources:

1. A TAM supplies the signed and encrypted TA binary and any required configuration data
2. A Client Application supplies the TA binary

This specification considers the first case where TA binary and configuration data are encrypted by recipient's public key that TAM

has to be involved. The second case will also be addressed separately.

2. Requirements Language

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 [RFC2119].

3. Terminology

3.1. Definitions

The definitions provided below are defined as used in this document. All the terms defined in the TEEP Architecture document [TEEPArch] will be used, which are not repeated in this document.

OTrP Broker: It is the Broker as defined in the TEEP Architecture document [TEEPArch].

3.2. Abbreviations

CA	Certificate Authority
OTrP	Open Trust Protocol
REE	Rich Execution Environment
SD	Security Domain
SP	Service Provider
TA	Trusted Application
TEE	Trusted Execution Environment
TFW	Trusted Firmware
TAM	Trusted Application Manager

4. OTrP Entities and Trust Model

4.1. System Components

The same system components as defined in the TEEP Architecture document [TEEPArch] are used in OTrP, including TAM, CA, TEE, REE, and OTrP Broker (a.k.a Broker).

Secure boot (for the purposes of OTrP) is optional in enabling authenticity checking of TEEs by the TAM. A TAM provider can choose its policy whether it trusts a TEE if the underlying firmware attestation information is not included.

OTrP uses trust anchors to establish trust between TEEs and TAMs and verifies that they communicate in a trusted way when performing lifecycle management transactions.

4.2. Trust Anchors in TEE

This assumes the Trust Anchor specification defined in the TEEP Architecture document [TEEPArch].

Each TEE comes with a trust store that contains a whitelist of root CA certificates that are used to validate a TAM's certificate. A TEE will accept a TAM to create new Security Domains and install new TAs on behalf of a SP only if the TAM's certificate is chained to one of the root CA certificates in the TEE's trust store.

4.3. Trust Anchors in TAM

The Trust Anchor set in a TAM consists of a list of Certificate Authority certificates that signs various device TEE certificates. A TAM decides what TEE and optionally TFW it will trust when TFW signature data is present in an attestation.

4.4. Keys and Certificate Types

OTrP leverages the following list of trust anchors and identities in generating signed and encrypted command messages that are exchanged between a device's TEE and a TAM. With these security artifacts, OTrP Messages are able to deliver end-to-end security without relying on any transport security.

Key Entity Name	Location	Issuer	Trust Implication	Cardinality
1. TFW key pair and certificate	Device secure storage	FW CA	A whitelist of FW root CA trusted by TAMs	1 per device
2. TEE key pair and certificate	Device TEE	TEE CA under a root CA	A whitelist of TEE root CA trusted by TAMs	1 per device
3. TAM key pair and certificate	TAM provider	TAM CA under a root CA	A whitelist of TAM root CA embedded in TEE	1 or multiple can be used by a TAM
4. SP key pair and certificate	SP	SP signer CA	TAM manages SP. TA trust is delegated to TAM. TEE trusts TAM to ensure that a TA is trustworthy.	1 or multiple can be used by a TAM

Table 1: Key and Certificate Types

1. TFW key pair and certificate: A key pair and certificate for evidence of trustworthy firmware in a device. This key pair is optional. Some TEE may present its trusted attributes to a TAM using signed attestation with a TFW key. For example, a platform that uses a hardware based TEE can have attestation data signed by a hardware protected TFW key.

Location: Device secure storage

Supported Key Type: RSA and ECC

Issuer: OEM CA

Trust Implication: A whitelist of FW root CA trusted by TAMs

Cardinality: One per device

2. TEE key pair and certificate: It is used for device attestation to a remote TAM and SP.

This key pair is burned into the device at device manufacturer. The key pair and its certificate are valid for the expected lifetime of the device.

Location: Device TEE

Supported Key Type: RSA and ECC

Issuer: A CA that chains to a TEE root CA

Trust Implication: A whitelist of TEE root CA trusted by TAMs

Cardinality: One per device

3. TAM key pair and certificate: A TAM provider acquires a certificate from a CA that a TEE trusts.

Location: TAM provider

Supported Key Type: RSA and ECC.

Supported Key Size: RSA 2048-bit, ECC P-256 and P-384. Other sizes should be anticipated in future.

Issuer: TAM CA that chains to a root CA

Trust Implication: A whitelist of TAM root CA embedded in TEE

Cardinality: One or multiple can be used by a TAM

4. SP key pair and certificate: an SP uses its own key pair and certificate to sign a TA.

Location: SP

Supported Key Type: RSA and ECC

Supported Key Size: RSA 2048-bit, ECC P-256 and P-384. Other sizes should be anticipated in future.

Issuer: an SP signer CA that chains to a root CA

Trust Implication: TAM manages SP. TA trusts an SP by validating trust against a TAM that the SP uses. A TEE trusts TAM to ensure that a TA from the TAM is trustworthy.

Cardinality: One or multiple can be used by an SP

5. Protocol Scope and Entity Relations

This document specifies messages and key properties that can establish mutual trust between a TEE and a TAM. The protocol provides specifications for the following three entities:

1. Key and certificate types required for device firmware, TEEs, TAs, SPs, and TAMs
2. Data message formats that should be exchanged between a TEE in a device and a TAM
3. An OTrP Broker in the REE that can relay messages between a Client Application and TEE

Figure 1: Protocol Scope and Entity Relationship

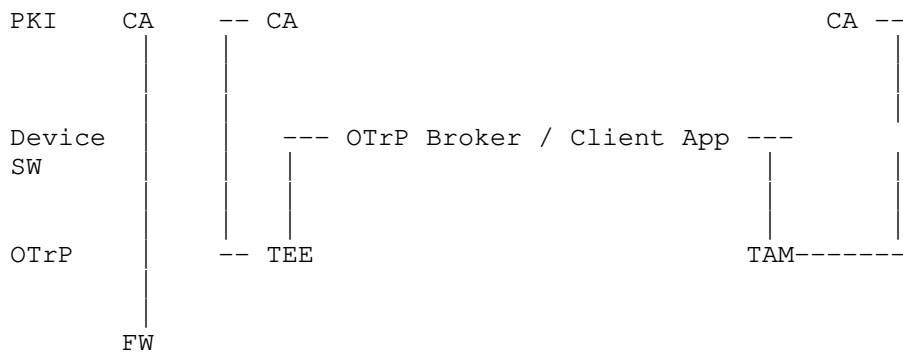
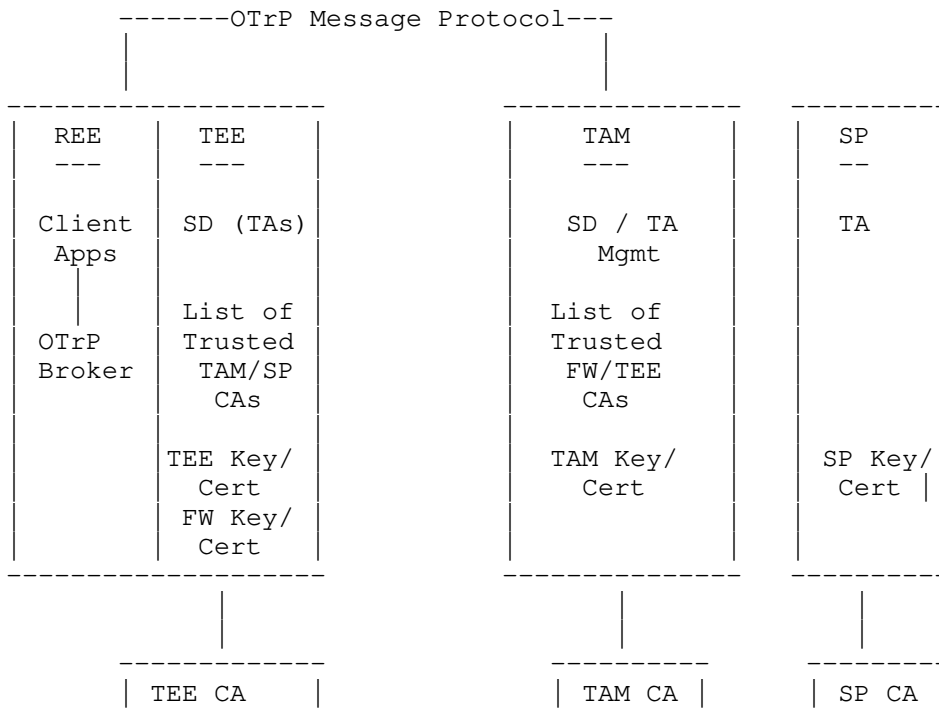


Figure 2: OTrP System Diagram



In the previous diagram, different Certificate Authorities can be used respectively for different types of certificates. OTrP Messages are always signed, where the signer keys is the message creator's private key such as a FW's private key, a TEE's private key, or a TAM's private key.

The main OTrP component consists of a set of standard JSON messages created by a TAM to deliver device SD and TA management commands to a device, and device attestation and response messages created by a TEE that responds to a TAM's OTrP message.

The communication method of OTrP Messages between a TAM and TEE in a device may vary between TAM and TEE providers. A mandatory transport protocol is specified for a compliant TAM and a device TEE.

An OTrP Broker is used to bridge communication between a TAM and a TEE. The OTrP Broker doesn't need to know the actual content of OTrP Messages except for the TEE routing information.

5.1. A Sample Device Setup Flow

Step 1: Prepare Images for Devices

1. [TEE vendor] Deliver TEE Image (CODE Binary) to device OEM
2. [CA] Deliver root CA Whitelist
3. [Soc] Deliver TFW Image

Step 2: Inject Key Pairs and Images to Devices

1. [OEM] Generate Secure Boot Key Pair (May be shared among multiple devices)
2. [OEM] Flash signed TFW Image and signed TEE Image onto devices (signed by Secure Boot Key)

Step 3: Setup attestation key pairs in devices

1. [OEM] Flash TFW Public Key and a bootloader key.
2. [TFW/TEE] Generate a unique attestation key pair and get a certificate for the device.

Step 4: Setup trust anchors in devices

1. [TFW/TEE] Store the key and certificate encrypted with the eFuse key
2. [TEE vendor or OEM] Store trusted CA certificate list into devices

5.2. Derived Keys in The Protocol

The protocol generates one key pair in run time to assist message communication and anonymous verification between a TAM and a TEE.

TEE SP Anonymous Key (AIK): one derived key pair per SP in a device

The purpose of the key pair is to sign data by a TEE without using its TEE device key for anonymous attestation to a Client Application. This key pair is generated in the first SD creation for an SP. It is deleted when all SDs are removed for a SP in a device. The public key of the key pair is given to the caller Client Application and TAM for future TEE returned data validation. The public key of this AIK is also used by a TAM to encrypt TA binary data and personalization data when it sends a TA to a device for installation.

5.3. Security Domain Hierarchy and Ownership

The primary job of a TAM is to help an SP to manage its trusted application components. A TA is typically installed in an SD. An SD is commonly created for an SP.

When an SP delegates its SD and TA management to a TAM, an SD is created on behalf of a TAM in a TEE and the owner of the SD is assigned to the TAM. An SD may be associated with an SP but the TAM has full privilege to manage the SD for the SP.

Each SD for an SP is associated with only one TAM. When an SP changes TAM, a new SP SD must be created to associate with the new TAM. The TEE will maintain a registry of TAM ID and SP SD ID mapping.

From an SD ownership perspective, the SD tree is flat and there is only one level. An SD is associated with its owner. It is up to TEE implementation how it maintains SD binding information for a TAM and different SPs under the same TAM.

It is an important decision in this protocol specification that a TEE doesn't need to know whether a TAM is authorized to manage the SD for an SP. This authorization is implicitly triggered by an SP Client Application, which instructs what TAM it wants to use. An SD is always associated with a TAM in addition to its SP ID. A rogue TAM isn't able to do anything on an unauthorized SP's SD managed by another TAM.

Since a TAM may support multiple SPs, sharing the same SD name for different SPs creates a dependency in deleting an SD. An SD can be deleted only after all TAs associated with this SD is deleted. An SP cannot delete a Security Domain on its own with a TAM if a TAM decides to introduce such sharing. There are cases where multiple virtual SPs belong to the same organization, and a TAM chooses to use the same SD name for those SPs. This is totally up to the TAM implementation and out of scope of this specification.

5.4. SD Owner Identification and TAM Certificate Requirements

There is a need of cryptographically binding proof about the owner of an SD in a device. When an SD is created on behalf of a TAM, a future request from the TAM must present itself as a way that the TEE can verify it is the true owner. The certificate itself cannot reliably be used as the owner because TAM may change its certificate.

To this end, each TAM will be associated with a trusted identifier defined as an attribute in the TAM certificate. This field is kept

the same when the TAM renew its certificates. A TAM CA is responsible to vet the requested TAM attribute value.

This identifier value must not collide among different TAM providers, and one TAM shouldn't be able to claim the identifier used by another TAM provider.

The certificate extension name to carry the identifier can initially use SubjectAltName:registeredID. A dedicated new extension name may be registered later.

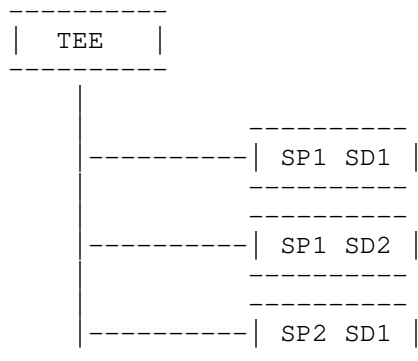
One common choice of the identifier value is the TAM's service URL. A CA can verify the domain ownership of the URL with the TAM in the certificate enrollment process.

A TEE can assign this certificate attribute value as the TAM owner ID for the SDs that are created for the TAM.

An alternative way to represent an SD ownership by a TAM is to have a unique secret key upon SD creation such that only the creator TAM is able to produce a Proof-of-Possession (POP) data with the secret.

5.5. Service Provider Container

A sample Security Domain hierarchy for the TEE is shown below.



OTrP segregates SDs and TAs such that a TAM can only manage or retrieve data for SDs and TAs that it previously created for the SPs it represents.

6. OTrP Broker

A TEE and TAs that run inside the TEE don't generally have capability to communicate to the outside of the hosting device, for example, the TEE specified by Global Platform groups [GPTEE]. This calls for a software module in the REE world to handle the network communication. Each Client Application in REE may carry this communication functionality but it must also interact with the TEE for the message exchange. The TEE interaction will vary according to different TEEs. In order for a Client Application to transparently support different TEEs, it is imperative to have a common interface for a Client Application to invoke for exchanging messages with TEEs.

A shared OTrP Broker comes to meet this need. An OTrP Broker is a Rich Application or SDK that facilitates communication between a TAM and TEE. It also provides interfaces for TAM SDK or Client Applications to query and trigger TA installation that the application needs to use.

This interface for Client Applications may be commonly an Android service call for an Android powered device. A Client Application interacts with a TAM, and turns around to pass messages received from TAM to OTrP Broker.

In all cases, a Client Application needs to be able to identify an OTrP Broker that it can use.

6.1. Role of OTrP Broker

An OTrP Broker abstracts the message exchanges with the TEE in a device. The input data is originated from a TAM that a Client Application connects. A Client Application may also directly call OTrP Broker for some TA query functions.

OTrP Broker may internally process a request from TAM. At least, it needs to know where to route a message, e.g. TEE instance. It doesn't need to process or verify message content.

OTrP Broker returns TEE / TFW generated response messages to the caller. OTrP Broker isn't expected to handle any network connection with an application or TAM.

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

6.2. OTrP Broker and Global Platform TEE Client API

A Client Application may use Global Platform (GP) TEE API for TA communication. OTrP may use the GP TEE Client API but it is internal to OTrP implementation that converts given messages from TAM. More details can be found at [GPTEECLAPI].

6.3. OTrP Broker Implementation Consideration

A Provider should consider methods of distribution, scope and concurrency on device and runtime options when implementing an OTrP Broker. Several non-exhaustive options are discussed below. Providers are encouraged to take advantage of the latest communication and platform capabilities to offer the best user experience.

6.3.1. OTrP Broker Distribution

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

It is important to ensure a legitimate OTrP Broker is installed and used. If an OTrP Broker is compromised it may send rogue messages to TAM and TEE and introduce additional risks.

6.3.2. Number of OTrP Broker

We anticipate only one shared OTrP Broker instance in a device. The device's TEE vendor will most probably supply one OTrP Broker. Potentially we expect some open source.

With one shared OTrP Broker, the OTrP Broker provider is responsible to allow multiple TAMs and TEE providers to achieve interoperability. With a standard OTrP Broker interface, TAM can implement its own SDK for its SP Client Applications to work with this OTrP Broker.

Multiple independent OTrP Broker providers can be used as long as they have standard interface to a Client Application or TAM SDK. Only one OTrP Broker is expected in a device.

TAM providers are generally expected to provide SDK for SP applications to interact with an OTrP Broker for the TAM and TEE interaction.

6.4. OTrP Broker Interfaces for Client Applications

A Client Application shall be responsible for relaying messages between the OTrP Broker and the TAM.

If a failure occurs during calling OTrP Broker, an error message described in "Common Errors" section (see Section 7.6) will be returned.

6.4.1. ProcessOTrPMessage call

Description

A Client Application will use this method of the OTrP Broker in a device to pass OTrP messages from a TAM. The method is responsible for interacting with the TEE and for forwarding the input message to the TEE. It also returns TEE generated response message back to the Client Application.

Inputs:

TAMInMsg - OTrP message generated in a TAM that is passed to this method from a Client Application.

Outputs:

A TEE-generated OTrP response message (which may be a successful response or be a response message containing an error raised within the TEE) for the client application to forward to the TAM. In the event of the OTrP Broker not being able to communicate with the TEE, a OTrPBrokerException shall be thrown.

6.4.2. GetTAInformation call

Description

A Client Application may quickly query local TEE about a previously installed TA without requiring TAM each time if it has had the TA's identifier and previously saved TEE SP AIK public key for TA information integrity verification.

Inputs:

```
{
  "TAQuery": {
    "spid": "<SP identifier value of the TA>",
    "taid": "<The identifier value of the TA>"
  }
}
```

Outputs:

The OTrP Broker is expected to return TA signer and TAM signer certificate along with other metadata information about the TA associated with the given identifier. It follows the underlying TEE trust model for authoring the local TA query from a Client Application.

The output is a JSON message that is generated by the TEE. It contains the following information:

- * tamid
- * SP ID
- * TA signer certificate
- * TAM certificate

The message is signed with TEE SP AIK private key.

The Client Application is expected to consume the response as follows.

The Client Application gets signed TA metadata, in particular, the TA signer certificate. It is able to verify that the result is from device by checking signer against TEE SP AIK public key it gets in some earlier interaction with TAM.

If this is a new Client Application in the device that hasn't had TEE SP AIK public key for the response verification, the application can contact the TAM first to do GetDeviceState, and TAM will return TEE SP AIK public key to the app for this operation to proceed.

Output Message:

```
{
  "TAInformationTBS": {
    "taid": "<TA Identifier from the input>",
    "tamid": "<TAM ID for the Security Domain where this TA
              resides>",
    "spid": "<The service provider identifier of this TA>",
    "signercert": "<The BASE64 encoded certificate data of the
                  TA binary application's signer certificate>",
    "signercacerts": [ < The full list of CA certificate chain
                        including the root CA>
    ],
    "cacert": "<The BASE64 encoded CA certificate data of the TA
              binary application's signer certificate>"
  ],
  "tamcert": "<The BASE64 encoded certificate data of the TAM
              that manages this TA.>",
  "tamcacerts": [ < The full list of CA certificate chain
                  including the root CA>
  ],
  "cacert": "<The BASE64 encoded CA certificate data of the TAM
            that manages this TA>"
  ]
}

{
  "TAInformation": {
    "payload": "<The BASE64URL encoding of the TAInformationTBS
               JSON above>",
    "protected": "<BASE64URL encoded signing algorithm>",
    "header": {
      "signer": { "<JWK definition of the TEE SP AIK public
                  key>" }
    },
    "signature": "<signature contents signed by TEE SP AIK
                  private key BASE64URL encoded>"
  }
}
```

where the definitions of BASE64 and BASE64URL refer to [RFC4648].

A sample JWK public key representation refers to an example in [RFC7517].

6.5. Sample End-to-End Client Application Flow

6.5.1. Case 1: A New Client Application Uses a TA

1. During the Client Application installation time, the Client Application calls TAM to initialize the device preparation step.
 - A. The Client Application knows it wants to use a Trusted Application TA1 but the application doesn't know whether TA1 has been installed or not. It can use GP TEE Client API [GPTEECLAPI] to check the existence of TA1 first. If it detects that TA1 doesn't exist, it will contact TAM to initiate the installation of TA1. Note that TA1 could have been previously installed by other Client Applications from the same service provider in the device.
 - B. The Client Application sends the TAM the TA list that it depends on. The TAM will query a device for the Security Domains and TAs that have been installed, and instructs the device to install any dependent TAs that have not been installed.
 - C. In general, the TAM has the latest TA list and their status in a device because all operations are instructed by TAM. TAM has such visibility because all Security Domain deletion and TA deletion are managed by the TAM; the TAM could have stored the state when a TA is installed, updated and deleted. There is also the possibility that an update command is carried out inside TEE but a response is never received in TAM. There is also possibility that some manual local reset is done in a device that the TAM isn't aware of the changes.
2. The TAM generates message: GetDeviceStateRequest
3. The Client Application passes the JSON message GetDeviceStateRequest to OTrP Broker call ProcessOTrPMessage. The communication between a Client Application and an OTrP Broker is up to the implementation of the OTrP Broker.
4. The OTrP Broker routes the message to the active TEE. Multiple TEE case: it is up to OTrP Broker to figure this out. This specification limits the support to only one active TEE, which is the typical case today.
5. The target active TEE processes the received OTrP message, and returns a JSON message GetDeviceStateResponse.

6. The OTrP Broker passes the GetDeviceStateResponse to the Client Application.
 7. The Client Application sends GetDeviceStateResponse to the TAM.
 8. The TAM processes the GetDeviceStateResponse.
 - A. Extract TEEspaik for the SP, signs TEEspaik with TAM signer key
 - B. Examine SD list and TA list
 9. The TAM continues to carry out other actions based on the need. The next call could be instructing the device to install a dependent TA.
 - A. Assume a dependent TA isn't in the device yet, the TAM may do the following: (1) create an SD in which to install the TA by sending a CreateSDRequest message. The message is sent back to the Client Application, and then the OTrP Broker and TEE to process; (2) install a TA with an InstallTARequest message.
 - B. If a Client Application depends on multiple TAs, the Client Application should expect multiple round trips of the TA installation message exchanges.
 10. At the last TAM and TEE operation, the TAM returns the signed TEE SP AIK public key to the application.
 11. The Client Application stores the TEEspaik for future loaded TA trust check.
 12. If the TAM finds that this is a fresh device that does not have any SD for the SP yet, then the TAM may next create an SD for the SP.
 13. During Client Application installation, the application checks whether required Trusted Applications are already installed, which may have been provided by the TEE. If needed, it will contact its TAM service to determine whether the device is ready or install TA list that this application needs.
- 6.5.2. Case 2: A Previously Installed Client Application Calls a TA
1. The Client Application checks the device readiness: (a) whether it has a TEE; (b) whether it has TA that it depends. It may happen that TAM has removed the TA this application depends on.

2. The Client Application calls the OTrP Broker to query the TA.
3. The OTrP Broker queries the TEE to get TA information. If the given TA doesn't exist, an error is returned.
4. The Client Application parses the TAInformation message.
5. If the TA doesn't exist, the Client Application calls its TAM to install the TA. If the TA exists, the Client Application proceeds to call the TA.

7. OTrP Messages

The main OTrP component is the set of standard JSON messages created by a TAM to deliver device SD and TA management commands to a device, and device attestation and response messages created by TEE to respond to TAM OTrP Messages.

An OTrP Message is designed to provide end-to-end security. It is always signed by its creator. In addition, an OTrP Message is typically encrypted such that only the targeted device TEE or TAM is able to decrypt and view the actual content.

7.1. Message Format

OTrP Messages use the JSON format for JSON's simple readability and moderate data size in comparison with alternative TLV and XML formats. More compact CBOR format may be used as an alternative choice.

JSON Message security has developed JSON Web Signing and JSON Web Encryption standard in the IETF Workgroup JOSE, see JWS [RFC7515] and JWE [RFC7516]. The OTrP Messages in this protocol will leverage the basic JWS and JWE to handle JSON signing and encryption.

7.2. Message Naming Convention

For each TAM command "xyz", OTrP use the following naming convention to represent its raw message content and complete request and response messages:

Purpose	Message Name	Example
Request to be signed	xyzTBSRequest	CreateSDTBSRequest
Request message	xyzRequest	CreateSDRequest
Response to be signed	xyzTBSResponse	CreateSDTBSResponse
Response message	xyzResponse	CreateSDResponse

7.3. Request and Response Message Template

An OTrP Request message uses the following format:

```
{
  "<name>TBSRequest": {
    <request message content>
  }
}
```

A corresponding OTrP Response message will be as follows.

```
{
  "<name>TBSResponse": {
    <response message content>
  }
}
```

7.4. Signed Request and Response Message Structure

A signed request message will generally include only one signature, and uses the flattened JWS JSON Serialization Syntax, see Section 7.2.2 in [RFC7515].

A general JWS object looks like the following.

```
{
  "payload": "<payload contents>",
  "protected": "<integrity-protected header contents>",
  "header": {
    <non-integrity-protected header contents>,
  },
  "signature": "<signature contents>"
}
```

OTrP signed messages only require the signing algorithm as the mandate header in the property "protected". The "non-integrity-protected header contents" is optional.

OTrP signed message will be given an explicit Request or Response property name. In other words, a signed Request or Response uses the following template.

A general JWS object looks like the following.

```
{
  "<name>[Request | Response]": {
    <JWS Message of <name>TBS[Request | Response]
  }
}
```

With the standard JWS message format, a signed OTrP Message looks like the following.

```
{
  "<name>[Request | Response]": {
    "payload": "<payload contents of <name>TBS[Request | Response]>",
    "protected": "<integrity-protected header contents>",
    "header": "<non-integrity-protected header contents>",
    "signature": "<signature contents>"
  }
}
```

The top element "<name>[Signed][Request|Response]" cannot be fully trusted to match the content because it doesn't participate in the signature generation. However, a recipient can always match it with the value associated with the property "payload". It purely serves to provide a quick reference for reading and method invocation.

Furthermore, most properties in an unsigned OTrP messages are encrypted to provide end-to-end confidentiality. The only OTrP message that isn't encrypted is the initial device query message that asks for the device state information.

Thus a typical OTrP Message consists of an encrypted and then signed JSON message. Some transaction data such as transaction ID and TEE information may need to be exposed to the OTrP Broker for routing purpose. Such information is excluded from JSON encryption. The device's signer certificate itself is encrypted. The overall final message is a standard signed JSON message.

As required by JSW/JWE, those JWE and JWS related elements will be BASE64URL encoded. Other binary data elements specific to the OTrP

specification are BASE64-encoded. This specification indicates elements that should be BASE64 and those elements that are to be BASE64URL encoded.

7.4.1. Identifying Signing and Encryption Keys for JWS/JWE Messaging

JWS and JWE messaging allow various options for identifying the signing and encryption keys, for example, it allows optional elements including "x5c", "x5t" and "kid" in the header to cover various possibilities.

To protect privacy, it is important that the device's certificate is released only to a trusted TAM, and that it is encrypted. The TAM will need to know the device certificate, but untrusted parties must not be able to get the device certificate. All OTrP messaging conversations between a TAM and device begin with `GetDeviceStateRequest` / `GetDeviceStateResponse`. These messages have elements built into them to exchange signing certificates, described in the section Section 9. Any subsequent messages in the conversation that follow on from this implicitly use the same certificates for signing/encryption, and as a result the certificates or references may be omitted in those subsequent messages.

In other words, the signing key identifier in the use of JWS and JWE here may be absent in the subsequent messages after the initial `GetDeviceState` query.

This has an implication on the TEE and TAM implementation: they have to cache the signer certificates for the subsequent message signature validation in the session. It may be easier for a TAM service to cache transaction session information but not so for a TEE in a device. A TAM can get a device's capability by checking the response message from a TEE to decide whether it should include its TAM signer certificate and OSCP data in each subsequent request message. The device's caching capability is reported in `GetDeviceStateResponse` `signerreq` parameter.

7.5. JSON Signing and Encryption Algorithms

The OTrP JSON signing algorithm shall use SHA256 or a stronger hash method with respective key type. JSON Web Algorithm RS256 or ES256 [RFC7518] SHALL be used for RSA with SHA256 and ECDSA with SHA256. If RSA with SHA256 is used, the JSON web algorithm representation is as follows.

```
{"alg": "RS256"}
```

The (BASE64URL encoded) "protected" header property in a signed message looks like the following:

```
"protected": "eyJhbGciOiJSUzI1NiJ9"
```

If ECSDA with P-256 curve and SHA256 are used for signing, the JSON signing algorithm representation is as follows.

```
{"alg": "ES256"}
```

The value for the "protected" field will be the following.

```
eyJhbGciOiJFUzI1NiJ9
```

Thus, a common OTrP signed message with ES256 looks like the following.

```
{
  "payload": "<payload contents>",
  "protected": "eyJhbGciOiJFUzI1NiJ9",
  "signature": "<signature contents>"
}
```

The OTrP JSON message encryption algorithm SHOULD use one of the supported algorithms defined in the later chapter of this document. JSON encryption uses a symmetric key as its "Content Encryption Key (CEK)". This CEK is encrypted or wrapped by a recipient's key. The OTrP recipient typically has an asymmetric key pair. Therefore, the CEK will be encrypted by the recipient's public key.

A compliant implementation shall support the following symmetric encryption algorithm and anticipate future new algorithms.

```
{"enc": "A128CBC-HS256"}
```

This algorithm represents encryption with AES 128 in CBC mode with HMAC SHA 256 for integrity. The value of the property "protected" in a JWE message will be

```
eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
```

An encrypted JSON message looks like the following.

```
{
  "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "recipients": [
    {
      "header": {
        "alg": "<RSA1_5 etc.>"
      },
      "encrypted_key": "<encrypted value of CEK>"
    }
  ],
  "iv": "<BASE64URL encoded IV data>",
  "ciphertext": "<Encrypted data over the JSON plaintext
    (BASE64URL)>",
  "tag": "<JWE authentication tag (BASE64URL)>"
}
```

OTrP doesn't use JWE AAD (Additional Authenticated Data) because each message is always signed after the message is encrypted.

7.5.1. Supported JSON Signing Algorithms

The following JSON signature algorithm is mandatory support in the TEE and TAM:

- o RS256

ES256 is optional to support.

7.5.2. Support JSON Encryption Algorithms

The following JSON authenticated encryption algorithm is mandatory support in TEE and TAM.

- o A128CBC-HS256

A256CBC-HS512 is optional to support.

7.5.3. Supported JSON Key Management Algorithms

The following JSON key management algorithm is mandatory support in TEE and TAM.

- o RSA1_5

ECDH-ES+A128KW and ECDH-ES+A256KW are optional to support.

7.6. Common Errors

An OTrP Response message typically needs to report the operation status and error causes if an operation fails. The following JSON message elements should be used across all OTrP Messages.

```
"status": "pass | fail"

"reason": {
  "error-code": "<error code if there is any>",
  "error-message": "<error message>"
}

"ver": "<version string>"
```

7.7. OTrP Message List

The following table lists the OTrP commands and therefore corresponding Request and Response messages defined in this specification. Additional messages may be added in the future when new task messages are needed.

GetDeviceState -

A TAM queries a device's current state with a message GetDeviceStateRequest. A device TEE will report its version, its FW version, and list of all SDs and TAs in the device that is managed by the requesting TAM. TAM may determine whether the device is trustworthy and decide to carry out additional commands according to the response from this query.

CreateSD -

A TAM instructs a device TEE to create an SD for an SP. The recipient TEE will check whether the requesting TAM is trustworthy.

UpdateSD -

A TAM instructs a device TEE to update an existing SD. A typical update need comes from SP certificate change, TAM certificate change and so on. The recipient TEE will verify whether the TAM is trustworthy and owns the SD.

DeleteSD -

A TAM instructs a device TEE to delete an existing SD. A TEE conditionally deletes TAs loaded in the SD according to a request parameter. An SD cannot be deleted until all TAs in this SD are deleted. If this is the last SD for an SP, TEE MAY also delete TEE SP AIK key for this SP.

InstallTA -

A TAM instructs a device to install a TA into an SD for a SP. The TEE in a device will check whether the TAM and TA are trustworthy.

UpdateTA -

A TAM instructs a device to update a TA into an SD for an SP. The change may commonly be bug fix for a previously installed TA.

DeleteTA -

A TAM instructs a device to delete a TA. The TEE in a device will check whether the TAM and TA are trustworthy.

7.8. OTrP Request Message Routing Rules

For each command that a TAM wants to send to a device, the TAM generates a request message. This is typically triggered by a Client Application that uses the TAM. The Client Application initiates contact with the TAM and receives TAM OTrP Request messages according to the TAM's implementation. The Client Application forwards the OTrP message to an OTrP Broker in the device, which in turn sends the message to the active TEE in the device.

The current version of this specification assumes that each device has only one active TEE, and the OTrP Broker is responsible to connect to the active TEE. This is the case today with devices in the market.

When the TEE responds to a request, the OTrP Broker gets the OTrP response messages back to the Client Application that sent the request. In case the target TEE fails to respond to the request, the OTrP Broker will be responsible to generate an error message to reply the Client Application. The Client Application forwards any data it received to its TAM.

7.8.1. SP Anonymous Attestation Key (SP AIK)

When the first new Security Domain is created in a TEE for an SP, a new key pair is generated and associated with this SP. This key pair is used for future device attestation to the service provider instead of using the device's TEE key pair.

8. Transport Protocol Support

The OTrP message exchange between a TEE device and TAM generally takes place between a Client Application in REE and TAM. A device that is capable to run a TEE and PKI based cryptographic attestation

isn't generally resource constraint to carry out standard HTTPS connections. A compliant device and TAM SHOULD support HTTPS.

9. Detailed Messages Specification

For each message in the following sections all JSON elements are mandatory if not explicitly indicated as optional.

9.1. GetDeviceState

This is the first command that a TAM will send to a device. This command is triggered when an SP's Client Application contacts its TAM to check whether the underlying device is ready for TA operations.

This command queries a device's current TEE state. A device TEE will report its version, its FW version, and list of all SDs and TAs in the device that is managed by the requesting TAM. TAM may determine whether the device is trustworthy and decide to carry out additional commands according to the response from this query.

The request message of this command is signed by the TAM. The response message from the TEE is encrypted. A random message encryption key (MK) is generated by TEE, and this encrypted key is encrypted by the TAM's public key such that only the TAM that sent the request is able to decrypt and view the response message.

9.1.1. GetDeviceStateRequest message

```
{
  "GetDeviceStateTBSRequest": {
    "ver": "1.0",
    "rid": "<Unique request ID>",
    "tid": "<transaction ID>",
    "ocspdats": [<a list of OCSP stapling data>],
    "supportedSigalgs": [<array of supported signing algorithms>]
  }
}
```

The request message consists of the following data elements:

ver - version of the message format

rid - a unique request ID generated by the TAM

tid - a unique transaction ID to trace request and response. This can be from a prior transaction's tid field, and can be used in subsequent message exchanges in this TAM session. The combination of rid and tid MUST be made unique.

ocspdat - A list of OCSP stapling data respectively for the TAM certificate and each of the CA certificates up to the root certificate. The TAM provides OCSP data such that a recipient TEE can validate the TAM certificate chain revocation status without making its own external OCSP service call. A TEE MAY cache the CA OCSP data such that the array may contain only the OCSP stapling data for the TAM certificate in subsequent exchanges. This is a mandatory field.

supportedsigalgs - an optional property to list the signing algorithms that the TAM is able to support. A recipient TEE MUST choose an algorithm in this list to sign its response message if this property is present in a request. If it is absent, the TEE may use any compliant signing algorithm that is listed as mandatory support in this specification.

The final request message is JSON signed message of the above raw JSON data with TAM's certificate.

```
{
  "GetDeviceStateRequest": {
    "payload": "<BASE64URL encoding of the GetDeviceStateTBSRequest
              JSON above>",
    "protected": "<BASE64URL encoded signing algorithm>",
    "header": {
      "x5c": "<BASE64 encoded TAM certificate chain up to the
            root CA certificate>"
    },
    "signature": "<signature contents signed by TAM private key>"
  }
}
```

The signing algorithm SHOULD use SHA256 with respective key type. The mandatory algorithm support is the RSA signing algorithm. The signer header "x5c" is used to include the TAM signer certificate up to the root CA certificate.

9.1.2. Request processing requirements at a TEE

Upon receiving a request message GetDeviceStateRequest at a TEE, the TEE MUST validate a request:

1. Validate JSON message signing. If it doesn't pass, an error message is returned.
2. Validate that the request TAM certificate is chained to a trusted CA that the TEE embeds as its trust anchor.

- * Cache the CA OCSP stapling data and certificate revocation check status for other subsequent requests.
- * A TEE can use its own clock time for the OCSP stapling data validation.

3. Optionally collect Firmware signed data

- * This is a capability in ARM architecture that allows a TEE to query Firmware to get FW signed data. It isn't required for all TEE implementations. When TFW signed data is absent, it is up to a TAM's policy how it will trust a TEE.

4. Collect SD information for the SD owned by this TAM

9.1.3. Firmware Signed Data

Firmware isn't expected to process or produce JSON data. It is expected to just sign some raw bytes of data.

The data to be signed by TFW key needs be some unique random data each time. The (UTF-8 encoded) "tid" value from the `GetDeviceStateTBSRequest` shall be signed by the firmware. TAM isn't expected to parse TFW data except the signature validation and signer trust path validation.

It is possible that a TEE can get some valid TFW signed data from another device. The TEE is responsible to validate TFW integrity to ensure that the underlying device firmware is trustworthy. In some cases, a TEE isn't able to get a TFW signed data, in which case the TEE trust validation is up to a TAM to decide. A TAM may opt to trust a TEE basing on the TEE signer and additional information about a TEE out-of-band.

When TFW signed data is available, a TAM validates the TEE and trusts that a trusted TEE has carried out appropriate trust check about a TFW.

```
TfwData: {  
  "tbs": "<TFW to be signed data, BASE64 encoded>",  
  "cert": "<BASE64 encoded TFW certificate>",  
  "sigalg": "Signing method",  
  "sig": "<TFW signed data, BASE64 encoded>"  
}
```

It is expected that a FW uses standard signature methods for maximal interoperability with TAM providers. The mandatory support list of signing algorithm is RSA with SHA256.

The JSON object above is constructed by a TEE with data returned from the FW. It isn't a standard JSON signed object. The signer information and data to be signed must be specially processed by a TAM according to the definition given here. The data to be signed is the raw data.

9.1.3.1. Supported Firmware Signature Methods

TAM providers shall support the following signature methods. A firmware provider can choose one of the methods in signature generation.

- o RSA with SHA256
- o ECDSA with SHA 256

The value of "sigalg" in the TfwData JSON message SHOULD use one of the following:

- o RS256
- o ES256

9.1.4. Post Conditions

Upon successful request validation, the TEE information is collected. There is no change in the TEE in the device.

The response message shall be encrypted where the encryption key shall be a symmetric key that is wrapped by TAM's public key. The JSON Content Encryption Key (CEK) is used for this purpose.

9.1.5. GetDeviceStateResponse Message

The message has the following structure.

```
{
  "GetDeviceTEESStateTBSResponse": {
    "ver": "1.0",
    "status": "pass | fail",
    "rid": "<the request ID from the request message>",
    "tid": "<the transaction ID from the request message>",
    "signerreq": true | false // about whether TAM needs to send
                        signer data again in subsequent messages,
    "edsi": "<Encrypted JSON DSI information>"
  }
}
```

where

signerreq - true if the TAM should send its signer certificate and OSCP data again in the subsequent messages. The value may be "false" if the TEE caches the TAM's signer certificate and OSCP status.

rid - the request ID from the request message

tid - the tid from the request message

edsi - the main data element whose value is JSON encrypted message over the following Device State Information (DSI).

The Device State Information (DSI) message consists of the following.

```

{
  "dsi": {
    "tfwdata": {
      "tbs": "<TFW to be signed data is the tid>",
      "cert": "<BASE64 encoded TFW certificate>",
      "sigalg": "Signing method",
      "sig": "<TFW signed data, BASE64 encoded>"
    },
    "tee": {
      "name": "<TEE name>",
      "ver": "<TEE version>",
      "cert": "<BASE64 encoded TEE cert>",
      "cacert": "<JSON array value of CA certificates up to
                  the root CA>",
      "sdlist": {
        "cnt": "<Number of SD owned by this TAM>",
        "sd": [
          {
            "name": "<SD name>",
            "spid": "<SP owner ID of this SD>",
            "talist": [
              {
                "taid": "<TA application identifier>",
                "taname": "<TA application friendly
                           name>" // optional
              }
            ]
          }
        ]
      },
      "teeaiklist": [
        {
          "spaik": "<SP AIK public key, BASE64 encoded>",
          "spaiktype": "<RSA | ECC>",
          "spid": "<sp id>"
        }
      ]
    }
  }
}

```

The encrypted JSON message looks like the following.

```
{
  "protected": "<BASE64URL encoding of encryption algorithm header
                JSON data>",
  "recipients": [
    {
      "header": {
        "alg": "RSA1_5"
      },
      "encrypted_key": "<encrypted value of CEK>"
    }
  ],
  "iv": "<BASE64URL encoded IV data>",
  "ciphertext": "<Encrypted data over the JSON object of dsi
                (BASE64URL)>",
  "tag": "<JWE authentication tag (BASE64URL)>"
}
```

Assume we encrypt plaintext with AES 128 in CBC mode with HMAC SHA 256 for integrity, the encryption algorithm header is:

```
{"enc": "A128CBC-HS256"}
```

The value of the property "protected" in the above JWE message will be

```
eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0
```

In other words, the above message looks like the following:

```
{
  "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "recipients": [
    {
      "header": {
        "alg": "RSA1_5"
      },
      "encrypted_key": "<encrypted value of CEK>"
    }
  ],
  "iv": "<BASE64URL encoded IV data>",
  "ciphertext": "<Encrypted data over the JSON object of dsi
                (BASE64URL)>",
  "tag": "<JWE authentication tag (BASE64URL)>"
}
```

The full response message looks like the following:


```

{
  "GetDeviceTEESStateTBSResponse": {
    "ver": "1.0",
    "status": "pass | fail",
    "rid": "<the request ID from the request message>",
    "tid": "<the transaction ID from the request message>",
    "signerreq": "true | false",
    "edsi": {
      "protected": "<BASE64URL encoding of encryption algorithm
                    header JSON data>",
      "recipients": [
        {
          "header": {
            "alg": "RSA1_5"
          },
          "encrypted_key": "<encrypted value of CEK>"
        }
      ],
      "iv": "<BASE64URL encoded IV data>",
      "ciphertext": "<Encrypted data over the JSON object of dsi
                    (BASE64URL)>",
      "tag": "<JWE authentication tag (BASE64URL)>"
    }
  }
}

```

The CEK will be encrypted by the TAM public key in the device. The TEE signed message has the following structure.

```

{
  "GetDeviceTEESStateResponse": {
    "payload": "<BASE64URL encoding of the JSON message
               GetDeviceTEESStateTBSResponse>",
    "protected": "<BASE64URL encoding of signing algorithm>",
    "signature": "<BASE64URL encoding of the signature value>"
  }
}

```

The signing algorithm shall use SHA256 with respective key type, see Section 7.5.1.

The final GetDeviceStateResponse response message consists of an array of TEE responses.

```
{
  "GetDeviceStateResponse": [ // JSON array
    {"GetDeviceTEEStateResponse": ...},
    ...
    {"GetDeviceTEEStateResponse": ...}
  ]
}
```

9.1.6. Error Conditions

An error may occur if a request isn't valid or the TEE runs into some error. The list of possible error conditions is the following.

ERR_REQUEST_INVALID The TEE meets the following conditions with a request message: (1) The request from a TAM has an invalid message structure; mandatory information is absent in the message; or an undefined member or structure is included. (2) TEE fails to verify the signature of the message or fails to decrypt its contents.

ERR_UNSUPPORTED_MSG_VERSION The TEE receives a version of message that the TEE can't deal with.

ERR_UNSUPPORTED_CRYPT_ALG The TEE receives a request message encoded with a cryptographic algorithm that the TEE doesn't support.

ERR_TFW_NOT_TRUSTED The TEE considers the underlying device firmware be not trustworthy.

ERR_TAM_NOT_TRUSTED The TEE needs to make sure whether the TAM is trustworthy by checking the validity of the TAM certificate and OSCP stapling data and so on. If the TEE finds the TAM is not reliable, it returns this error code.

ERR_TEE_FAIL If the TEE fails to process a request because of its internal error but is able to sign an error response message, it will return this error code.

The response message will look like the following if the TEE signing can work to sign the error response message.

```
{
  "GetDeviceTEEStateTBSResponse": {
    "ver": "1.0",
    "status": "fail",
    "rid": "<the request ID from the request message>",
    "tid": "<the transaction ID from the request message>",
    "reason": {"error-code": "<error code>"}
    "supportedsigalgs": [<an array of signature algorithms that
                        the TEE supports>]
  }
}
```

where

supportedsigalgs - an optional property to list the JWS signing algorithms that the active TEE supports. When a TAM sends a signed message that the TEE isn't able to validate, it can include signature algorithms that it is able to consume in this status report. A TAM can generate a new request message to retry the management task with a TEE-supported signing algorithm.

If the TEE isn't able to sign an error message due to an internal device error, a general error message should be returned by the OTrP Broker.

9.1.7. TAM Processing Requirements

Upon receiving a GetDeviceStateResponse message at a TAM, the TAM MUST validate the following.

- o Parse to get list of GetDeviceTEEStateResponse JSON objects
- o Parse the JSON "payload" property and decrypt the JSON element "edsi". The decrypted message contains the TEE signer certificate.
- o Validate the GetDeviceTEEStateResponse JSON signature. The signer certificate is extracted from the decrypted message in the last step.
- o Extract TEE information and check it against its TEE acceptance policy.
- o Extract the TFW signed element, and check the signer and data integration against its TFW policy.
- o Check the SD list and TA list and prepare for a subsequent command such as "CreateSD" if it needs to have a new SD for an SP.

9.2. Security Domain Management

9.2.1. CreateSD

This command is typically preceded with a `GetDeviceState` command that has acquired the device information of the target device by the TAM. The TAM sends such a command to instruct a TEE to create a new Security Domain for an SP.

A TAM sends an OTrP `CreateSDRequest` Request message to a device TEE to create a Security Domain for an SP. Such a request is signed by the TAM where the TAM signer may or may not be the same as the SP's TA signer certificate. The resulting SD is associated with two identifiers for future management:

- o TAM as the owner. The owner identifier is a registered unique TAM ID that is stored in the TAM certificate.
- o SP identified by its TA signer certificate as the authorization. A TAM can add more than one SP certificate to an SD.

A Trusted Application that is signed by a matching SP signer certificate for an SD is eligible to be installed into that SD. The TA installation into an SD by a subsequent `InstallTARrequest` message may be instructed from a TAM.

9.2.1.1. CreateSDRequest Message

The request message for CreateSD has the following JSON format.

```
{
  "CreateSDTBSRequest": {
    "ver": "1.0",
    "rid": "<unique request ID>",
    "tid": "<transaction ID>", // this may be from prior message
    "tee": "<TEE routing name from the DSI for the SD's target>",
    "nextdsi": true | false,
    "dsihash": "<hash of DSI returned in the prior query>",
    "content": ENCRYPTED { // this piece of JSON data will be
                        // encrypted
      "spid": "<SP ID value>",
      "sdname": "<SD name for the domain to be created>",
      "spcert": "<BASE64 encoded SP certificate>",
      "tamid": "<An identifiable attribute of the TAM
                certificate>",
      "did": "<SHA256 hash of the TEE cert>"
    }
  }
}
```

In the message,

rid - A unique value to identify this request

tid - A unique value to identify this transaction. It can have the same value for the tid in the preceding GetDeviceStateRequest.

tee - TEE ID returned from the previous GetDeviceStateResponse.

nextdsi - Indicates whether the up-to-date Device State Information (DSI) is expected in the response from the TEE to this request.

dsihash - The BASE64-encoded SHA256 hash value of the DSI data returned in the prior TAM operation with this target TEE. This value is always included such that a receiving TEE can check whether the device state has changed since its last query. It helps enforce SD update order in the right sequence without accidentally overwriting an update that was done simultaneously.

content - The "content" is a JSON encrypted message that includes actual input for the SD creation. The encryption key is TAMmk that is encrypted by the target TEE's public key. The entire message is signed by the TAM private key TAMpriv. A separate TAMmk isn't used in the latest specification because JSON encryption will use a content encryption key for exactly the same purpose.

spid - A unique id assigned by the TAM for its SP. It should be unique within a TAM namespace.

sdname - a name unique to the SP. TAM should ensure it is unique for each SP.

spcert - The SP's TA signer certificate is included in the request. This certificate will be stored by the device TEE which uses it to check against TA installation. Only if a TA is signed by a matching spcert associated with an SD will the TA be installed into the SD.

tamid - SD owner claim by TAM - an SD owned by a TAM will be associated with a trusted identifier defined as an attribute in the signer TAM certificate. TEE will be responsible to assign this ID to the SD. The TAM certificate attribute for this attribute tamid MUST be vetted by the TAM signer issuing CA. With this trusted identifier, the SD query at TEE can be fast upon TAM signer verification.

did - The SHA256 hash of the binary-encoded device TEE certificate. The encryption key CEK will be encrypted the recipient TEE's public key. This hash value in the "did" property allows the recipient TEE to check whether it is the expected target to receive such a request. If this isn't given, an OTrP message for device 2 could be sent to device 1. It is optional for the TEE to check because the successful decryption of the request message with this device's TEE private key already proves it is the target. This explicit hash value makes the protocol not dependent on message encryption method in future.

A CreateSDTBSRequest message is signed to generate a final CreateSDRequest message as follows.

```
{
  "CreateSDRequest": {
    "payload": "<CreateSDTBSRequest JSON above>",
    "protected": "<integrity-protected header contents>",
    "header": "<non-integrity-protected header contents>",
    "signature": "<signature contents signed by TAM private key>"
  }
}
```

The TAM signer certificate is included in the "header" property.

9.2.1.2. Request Processing Requirements at a TEE

Upon receiving a CreateSDRequest request message at a TEE, the TEE MUST do the following:

1. Validate the JSON request message as follows
 - * Validate JSON message signing.
 - * Validate that the request TAM certificate is chained to a trusted CA that the TEE embeds as its trust anchor.
 - * Compare dsihash with its current state to make sure nothing has changed since this request was sent.
 - * Decrypt to get the plaintext of the content: (a) spid, (b) sd name, (c) did.
 - * Check that an SPID is supplied.
 - * spcert check: check it is a valid certificate (signature and format verification only).
 - * Check "did" is the SHA256 hash of its TEEcert BER raw binary data.
 - * Check whether the requested SD already exists for the SP.
 - * Check that the tamid in the request matches the TAM certificate's TAM ID attribute.
2. If the request was valid, create action
 - * Create an SD for the SP with the given name.
 - * Assign the tamid from the TAMCert to this SD.
 - * Assign the SPID and SPCert to this SD.
 - * Check whether a TEE SP AIK key pair already exists for the given SP ID.
 - * Create TEE SP AIK key pair if it doesn't exist for the given SP ID.
 - * Generate new DSI data if the request asks for updated DSI.
3. Construct a CreateSDResponse message

- * Create raw content
 - + Operation status
 - + "did" or full signer certificate information,
 - + TEE SP AIK public key if DSI isn't going to be included
 - + Updated DSI data if requested
 - * The response message is encrypted with the same JWE CEK of the request without recreating a new content encryption key.
 - * The encrypted message is signed with TEEpriv. The signer information ("did" or TEEcert) is encrypted.
4. Deliver the response message. (a) The OTrP Broker returns this to the Client Application; (b) The Client App passes this back to the TAM.
 5. TAM processing. (a) The TAM processes the response message; (b) the TAM can look up signer certificate from the device ID "did".

If a request is illegitimate or signature doesn't pass, a "status" property in the response will indicate the error code and cause.

9.2.1.3. CreateSDResponse Message

The response message for a CreateSDRequest contains the following content.

```
{
  "CreateSDTBSResponse": {
    "ver": "1.0",
    "status": "<operation result>",
    "rid": "<the request ID received>",
    "tid": "<the transaction ID received>",
    "content": ENCRYPTED {
      "reason": "<failure reason detail>", // optional
      "did": "<the device id received from the request>",
      "sdname": "<SD name for the domain created>",
      "teespaik": "<TEE SP AIK public key, BASE64 encoded>",
      "dsi": "<Updated TEE state, including all SDs owned by
        this TAM>"
    }
  }
}
```


In the response message, the following fields MUST be supplied.

did - The SHA256 hash of the device TEE certificate. This shows the device ID explicitly to the receiving TAM.

teespaik - The newly generated SP AIK public key for the given SP. This is an optional value if the device has had another domain for the SP that has triggered TEE SP AIK key pair for this specific SP.

There is a possible extreme error case where the TEE isn't reachable or the TEE final response generation itself fails. In this case, the TAM might still receive a response from the OTrP Broker if the OTrP Broker is able to detect such error from TEE. In this case, a general error response message should be returned by the OTrP Broker, assuming OTrP Broker even doesn't know any content and information about the request message.

In other words, the TAM should expect to receive a TEE successfully signed JSON message, a general "status" message, or none when a client experiences a network error.

```
{
  "CreateSDResponse": {
    "payload": "<CreateSDTBSResponse JSON above>",
    "protected": {
      "<BASE64URL of signing algorithm>"
    },
    "signature": "<signature contents signed by the TEE device private
                  key (BASE64URL)>"
  }
}
```

When the TEE fails to respond, the OTrP Broker will not provide a subsequent response to the TAM. The TAM should treat this as if the device has gone offline where a response is never delivered back.

9.2.1.4. Error Conditions

An error might occur if a request isn't valid or the TEE runs into some error. The list of possible errors are as follows. Refer to the Error Code List (Section 13.1) for detailed causes and actions.

ERR_REQUEST_INVALID

ERR_UNSUPPORTED_MSG_VERSION

ERR_UNSUPPORTED_CRYPTO_ALG

ERR_DEV_STATE_MISMATCH

ERR_SD_ALREADY_EXIST

ERR_SD_NOT_FOUND

ERR_SPCERT_INVALID

ERR_TEE_FAIL

ERR_TAM_NOT_AUTHORIZED

ERR_TAM_NOT_TRUSTED

9.2.2. UpdateSD

This TAM initiated command can update an SP's SD that it manages for any of the following needs: (a) Update an SP signer certificate; (b) Add an SP signer certificate when an SP uses multiple to sign TA binaries; (c) Update an SP ID.

The TAM presents the proof of the SD ownership to the TEE, and includes related information in its signed message. The entire request is also encrypted for end-to-end confidentiality.

9.2.2.1. UpdateSDRequest Message

The UpdateSD request message has the following JSON format.

```
{
  "UpdateSDTBSRequest": {
    "ver": "1.0",
    "rid": "<unique request ID>",
    "tid": "<transaction ID>", // this may be from prior message
    "tee": "<TEE routing name from the DSI for the SD's target>",
    "nextdsi": true | false,
    "dsihash": "<hash of DSI returned in the prior query>",
    "content": ENCRYPTED { // this piece of JSON will be encrypted
      "tamid": "<tamid associated with this SD>",
      "spid": "<SP ID>",
      "sdname": "<SD name for the domain to be updated>",
      "changes": {
        "newsdname": "<Change the SD name to this new name>",
          // Optional
        "newspid": "<Change SP ID of the domain to this new value>",
          // Optional
        "spcert": ["<BASE64 encoded new SP signer cert to be added>"],
          // Optional
        "deloldspcert": ["<The SHA256 hex value of an old SP cert
          assigned into this SD that should be deleted >"],
          // Optional
        "renewteespaik": true | false
      }
    }
  }
}
```

In the message,

rid - A unique value to identify this request

tid - A unique value to identify this transaction. It can have the same value as the tid in the preceding GetDeviceStateRequest.

tee - TEE ID returned from the previous GetDeviceStateResponse

nextdsi - Indicates whether the up-to-date Device State Information (DSI) is expected to be returned in the response from the TEE to this request.

dsihash - The BASE64-encoded SHA256 hash value of the DSI data returned in the prior TAM operation with this target TEE. This value is always included such that a receiving TEE can check whether the device state has changed since its last query. It

helps enforce SD update order in the right sequence without accidentally overwriting an update that was done simultaneously.

content - The "content" is a JSON encrypted message that includes actual input for the SD update. The standard JSON content encryption key (CEK) is used, and the CEK is encrypted by the target TEE's public key.

tamid - SD owner claim by TAM - an SD owned by a TAM will be associated with a trusted identifier defined as an attribute in the signer TAM certificate.

spid - the identifier of the SP whose SD will be updated. This value is still needed because the SD name is considered unique only within an SP.

sdname - the name of the target SD to be updated.

changes - its content consists of changes are to be updated in the given SD.

newsdname - the new name of the target SD to be assigned if this value is present.

newspid - the new SP ID of the target SD to be assigned if this value is present.

spcert - a new TA signer certificate of this SP to be added to the SD if this is present.

deloldspcert - an SP certificate assigned into the SD is to be deleted if this is present. The value is the SHA256 fingerprint of the old SP certificate.

renewteespaik - the value should be true or false. If it is present and the value is true, the TEE MUST regenerate TEE SP AIK for this SD's owner SP. The newly generated TEE SP AIK for the SP must be returned in the response message of this request. If there is more than one SD for the SP, a new SPID for one of the domains will always trigger a new teespaik generation as if a new SP were introduced to the TEE.

The UpdateSDTBSRequest message is signed to generate the final UpdateSDRequest message.

```
{
  "UpdateSDRequest": {
    "payload": "<UpdateSDTBSRequest JSON above>",
    "protected": "<integrity-protected header contents>",
    "header": "<non-integrity-protected header contents>",
    "signature": "<signature contents signed by TAM private key>"
  }
}
```

TAM signer certificate is included in the "header" property.

9.2.2.2. Request Processing Requirements at a TEE

Upon receiving a request message UpdateSDRequest at a TEE, the TEE must validate a request:

1. Validate the JSON request message

- * Validate JSON message signing
- * Validate that the request TAM certificate is chained to a trusted CA that the TEE embeds as its trust anchor. The TAM certificate status check is generally not needed anymore in this request. The prior request should have validated the TAM certificate's revocation status.
- * Compare dsihash with the TEE cached last response DSI data to this TAM.
- * Decrypt to get the plaintext of the content.
- * Check that the target SD name is supplied.
- * Check whether the requested SD exists.
- * Check that the TAM owns this TAM by verifying tamid in the SD matches TAM certificate's TAM ID attribute.
- * Now the TEE is ready to carry out update listed in the "content" message.

2. If the request is valid, update action

- * If "newsdname" is given, replace the SD name for the SD to the new value

- * If "newspid" is given, replace the SP ID assigned to this SD with the given new value
 - * If "spcert" is given, add this new SP certificate to the SD.
 - * If "deloldspcert" is present in the content, check previously assigned SP certificates to this SD, and delete the one that matches the given certificate hash value.
 - * If "renewteespaik" is given and has a value of 'true', generate a new TEE SP AIK key pair, and replace the old one with this.
 - * Generate new DSI data if the request asks for updated DSI
 - * Now the TEE is ready to construct the response message
3. Construct UpdateSDResponse message
- * Create raw content
 - + Operation status
 - + "did" or full signer certificate information,
 - + TEE SP AIK public key if DSI isn't going to be included
 - + Updated DSI data if requested
 - * The response message is encrypted with the same JWE CEK of the request without recreating a new content encryption key.
 - * The encrypted message is signed with TEEpriv. The signer information ("did" or TEEcert) is encrypted.
4. Deliver response message. (a) The OTrP Broker returns this to the app; (b) The app passes this back to the TAM.
5. TAM processing. (a) The TAM processes the response message; (b) The TAM can look up the signer certificate from the device ID "did".

If a request is illegitimate or the signature doesn't pass, a "status" property in the response will indicate the error code and cause.

9.2.2.3. UpdateSDResponse Message

The response message for a UpdateSDRequest contains the following content.

```
{
  "UpdateSDTBSResponse": {
    "ver": "1.0",
    "status": "<operation result>",
    "rid": "<the request ID received>",
    "tid": "<the transaction ID received>",
    "content": ENCRYPTED {
      "reason": "<failure reason detail>", // optional
      "did": "<the device id hash>",
      "cert": "<TEE certificate>", // optional
      "teespaik": "<TEE SP AIK public key, BASE64 encoded>",
      "teespaiktype": "<TEE SP AIK key type: RSA or ECC>",
      "dsi": "<Updated TEE state, including all SD owned by
        this TAM>"
    }
  }
}
```

In the response message, the following fields MUST be supplied.

did - The request should have known the signer certificate of this device from a prior request. This hash value of the device TEE certificate serves as a quick identifier only. A full device certificate isn't necessary.

teespaik - the newly generated SP AIK public key for the given SP if the TEE SP AIK for the SP is asked to be renewed in the request. This is an optional value if "dsi" is included in the response, which will contain all up-to-date TEE SP AIK key pairs.

Similar to the template for the creation of the encrypted and signed CreateSDResponse, the final UpdateSDResponse looks like the following.

```
{
  "UpdateSDResponse": {
    "payload": "<UpdateSDTBSResponse JSON above>",
    "protected": {
      "<BASE64URL of signing algorithm>"
    },
    "signature": "<signature contents signed by TEE device private
                  key (BASE64URL)>"
  }
}
```

When the TEE fails to respond, the OTrP Broker will not provide a subsequent response to the TAM. The TAM should treat this as if the device has gone offline where a response is never delivered back.

9.2.2.4. Error Conditions

An error may occur if a request isn't valid or the TEE runs into some error. The list of possible errors are as follows. Refer to the Error Code List (Section 13.1) for detailed causes and actions.

ERR_REQUEST_INVALID

ERR_UNSUPPORTED_MSG_VERSION

ERR_UNSUPPORTED_CRYPTO_ALG

ERR_DEV_STATE_MISMATCH

ERR_SD_NOT_FOUND

ERR_SDNAME_ALREADY_USED

ERR_SPCERT_INVALID

ERR_TEE_FAIL

ERR_TAM_NOT_AUTHORIZED

ERR_TAM_NOT_TRUSTED

9.2.3. DeleteSD

A TAM sends a DeleteSDRequest message to a TEE to delete a specified SD that it owns. An SD can be deleted only if there is no TA associated with this SD in the device. The request message can

contain a flag to instruct the TEE to delete all related TAs in an SD and then delete the SD.

The target TEE will operate with the following logic.

1. Look up the given SD specified in the request message
2. Check that the TAM owns the SD
3. Check that the device state hasn't changed since the last operation
4. Check whether there are TAs in this SD
5. If TA exists in an SD, check whether the request instructs whether the TA should be deleted. If the request instructs the TEE to delete TAs, delete all TAs in this SD. If the request doesn't instruct the TEE to delete TAs, return an error "ERR_SD_NOT_EMPTY".
6. Delete the SD
7. If this is the last SD of this SP, delete the TEE SP AIK key.

9.2.3.1. DeleteSDRequest Message

The request message for DeleteSD has the following JSON format.

```
{
  "DeleteSDTBSRequest": {
    "ver": "1.0",
    "rid": "<unique request ID>",
    "tid": "<transaction ID>", // this may be from prior message
    "tee": "<TEE routing name from the DSI for the SD's target>",
    "nextdsi": true | false,
    "dsihash": "<hash of DSI returned in the prior query>",
    "content": ENCRYPTED { // this piece of JSON will be encrypted
      "tamid": "<tamid associated with this SD>",
      "sdname": "<SD name for the domain to be updated>",
      "deleteta": true | false
    }
  }
}
```

In the message,

rid - A unique value to identify this request

tid - A unique value to identify this transaction. It can have the same value for the tid in the preceding GetDeviceStateRequest.

tee - TEE ID returned from the previous response
GetDeviceStateResponse

nextdsi - Indicates whether the up-to-date Device State Information (DSI) is to be returned in the response to this request.

dsihash - The BASE64-encoded SHA256 hash value of the DSI data returned in the prior TAM operation with this target TEE. This value is always included such that a receiving TEE can check whether the device state has changed since its last query. It helps enforce SD update order in the right sequence without accidentally overwriting an update that was done simultaneously.

content - The "content" is a JSON encrypted message that includes actual input for the SD update. The standard JSON content encryption key (CEK) is used, and the CEK is encrypted by the target TEE's public key.

tamid - SD owner claim by TAM - an SD owned by a TAM will be associated with a trusted identifier defined as an attribute in the signer TAM certificate.

sdname - the name of the target SD to be updated.

deleteta - the value should be boolean 'true' or 'false'. If it is present and the value is 'true', the TEE should delete all TAs associated with the SD in the device.

According to the OTrP message template, the full request DeleteSDRequest is a signed message over the DeleteSDTBSRequest as follows.

```
{
  "DeleteSDRequest": {
    "payload": "<DeleteSDTBSRequest JSON above>",
    "protected": "<integrity-protected header contents>",
    "header": "<non-integrity-protected header contents>",
    "signature": "<signature contents signed by TAM private key>"
  }
}
```

TAM signer certificate is included in the "header" property.

9.2.3.2. Request Processing Requirements at a TEE

Upon receiving a request message DeleteSDRequest at a TEE, the TEE must validate a request:

1. Validate the JSON request message
 - * Validate JSON message signing
 - * Validate that the request TAM certificate is chained to a trusted CA that the TEE embeds as its trust anchor. The TAM certificate status check is generally not needed anymore in this request. The prior request should have validated the TAM certificate's revocation status.
 - * Compare dsihash with the TEE cached last response DSI data to this TAM
 - * Decrypt to get the plaintext of the content
 - * Check that the target SD name is supplied
 - * Check whether the requested SD exists
 - * Check that the TAM owns this TAM by verifying that the tamid in the SD matches the TAM certificate's TAM ID attribute
 - * Now the TEE is ready to carry out the update listed in the "content" message
2. If the request is valid, deletion action
 - * Check TA existence in this SD
 - * If "deleteta" is "true", delete all TAs in this SD. If the value of "deleteta" is false and some TA exists, return an error "ERR_SD_NOT_EMPTY"
 - * Delete the SD
 - * Delete the TEE SP AIK key pair if this SD is the last one for the SP
 - * Now the TEE is ready to construct the response message
3. Construct a DeleteSDResponse message
 - * Create response content

- + Operation status
 - + "did" or full signer certificate information,
 - + Updated DSI data if requested
- * The response message is encrypted with the same JWE CEK of the request without recreating a new content encryption key.
 - * The encrypted message is signed with TEEpriv. The signer information ("did" or TEEcert) is encrypted.
4. Deliver response message. (a) The OTrP Broker returns this to the app; (b) The app passes this back to the TAM
 5. TAM processing. (a) The TAM processes the response message; (b) The TAM can look up signer certificate from the device ID "did".

If a request is illegitimate or the signature doesn't pass, a "status" property in the response will indicate the error code and cause.

9.2.3.3. DeleteSDResponse Message

The response message for a DeleteSDRequest contains the following content.

```
{
  "DeleteSDTBSResponse": {
    "ver": "1.0",
    "status": "<operation result>",
    "rid": "<the request ID received>",
    "tid": "<the transaction ID received>",
    "content": ENCRYPTED {
      "reason": "<failure reason detail>", // optional
      "did": "<the device id hash>",
      "dsi": "<Updated TEE state, including all SD owned by
             this TAM>"
    }
  }
}
```

In the response message, the following fields MUST be supplied.

did - The request should have known the signer certificate of this device from a prior request. This hash value of the device TEE certificate serves as a quick identifier only. A full device certificate isn't necessary.

The final DeleteSDResponse looks like the following.

```
{
  "DeleteSDResponse": {
    "payload": "<DeleteSDTBSResponse JSON above>",
    "protected": {
      "<BASE64URL of signing algorithm>"
    },
    "signature": "<signature contents signed by TEE device
      private key (BASE64URL)>"
  }
}
```

When the TEE fails to respond, the OTrP Broker will not provide a subsequent response to the TAM. The TAM should treat this as if the device has gone offline where a response is never delivered back.

9.2.3.4. Error Conditions

An error may occur if a request isn't valid or the TEE runs into some error. The list of possible errors is as follows. Refer to the Error Code List (Section 13.1) for detailed causes and actions.

ERR_REQUEST_INVALID

ERR_UNSUPPORTED_MSG_VERSION

ERR_UNSUPPORTED_CRYPTO_ALG

ERR_DEV_STATE_MISMATCH

ERR_SD_NOT_EMPTY

ERR_SD_NOT_FOUND

ERR_TEE_FAIL

ERR_TAM_NOT_AUTHORIZED

ERR_TAM_NOT_TRUSTED

9.3. Trusted Application Management

This protocol doesn't introduce a TA container concept. All TA authorization and management will be up to the TEE implementation.

The following three TA management commands are supported.

- o InstallTA - provision a TA by TAM
- o UpdateTA - update a TA by TAM
- o DeleteTA - remove TA registration information with an SD, remove the TA binary and all TA-related data in a TEE

9.3.1. InstallTA

TA binary data and related personalization data if there is any can be from two sources:

1. A TAM supplies the signed and encrypted TA binary
2. A Client Application supplies the TA binary

This specification primarily considers the first case where a TAM supplies a TA binary. This is to ensure that a TEE can properly validate whether a TA is trustworthy. Further, TA personalization data will be encrypted by the TEE device's SP public key for end-to-end protection. A Client Application bundled TA case will be addressed separately later.

A TAM sends the following information in a InstallTAREquest message to a target TEE:

- o The target SD information: SP ID and SD name
- o Encrypted TA binary data. TA data is encrypted with the TEE SP AIK.
- o TA metadata. It is optional to include the SP signer certificate for the SD to add if the SP has changed signer since the SD was created.

The TEE processes the command given by the TAM to install a TA into an SP's SD. It does the following:

- o Validation
 - * The TEE validates the TAM message authenticity
 - * Decrypt to get request content
 - * Look up the SD with the SD name
 - * Checks that the TAM owns the SD

- * Checks that the DSI hash matches which shows that the device state hasn't changed
- o If the request is valid, continue to do the TA validation
 - * Decrypt to get the TA binary data and any personalization data with the "TEE SP AIK private key"
 - * Check that SP ID is the one that is registered with the SP SD
 - * Check that the TA signer is either a newly given SP certificate or the one that is already trusted by the SD from the previous TA installation. The TA signing method is specific to a TEE. This specification doesn't define how a TA should be signed; a TAM should support TEE specific TA signing when it supports that TEE.
 - * If a TA signer is given in the request, add this signer into the SD.
- o If the above validation passed, continue to do TA installation
 - * The TEE re-encrypts the TA binary and its personalization data with its own method.
 - * The TEE enrolls and stores the TA in a secure storage.
- o Construct a response message. This involves signing encrypted status information for the requesting TAM.

9.3.1.1. InstallTAResponse Message

The request message for InstallTA has the following JSON format.

```
{
  "InstallTATBSRequest": {
    "ver": "1.0",
    "rid": "<unique request ID>",
    "tid": "<transaction ID>",
    "tee": "<TEE routing name from the DSI for the SD's target>",
    "nextdsi": true | false,
    "dsihash": "<hash of DSI returned in the prior query>",
    "content": ENCRYPTED {
      "tamid": "<TAM ID previously assigned to the SD>",
      "spid": "<SPID value>",
      "sdname": "<SD name for the domain to install the TA>",
      "spcert": "<BASE64 encoded SP certificate >", // optional
      "taid": "<TA identifier>"
    },
    "encrypted_ta": {
      "key": "<JWE enveloped data of a 256-bit symmetric key by
        the recipient's TEEspaik public key>",
      "iv": "<hex of 16 random bytes>",
      "alg": "<encryption algorithm. AESCBC by default.>",
      "ciphertadata": "<BASE64 encoded encrypted TA binary data>",
      "cipherpdata": "<BASE64 encoded encrypted TA personalization
        data>"
    }
  }
}
```

In the message,

rid - A unique value to identify this request

tid - A unique value to identify this transaction. It can have the same value for the tid in the preceding GetDeviceStateRequest.

tee - TEE ID returned from the previous GetDeviceStateResponse

nextdsi - Indicates whether the up-to-date Device State Information (DSI) is to be returned in the response to this request.

dsihash - The BASE64-encoded SHA256 hash value of the DSI data returned in the prior TAM operation with this target TEE. This value is always included such that a receiving TEE can check whether the device state has changed since its last query. It helps enforce SD update order in the right sequence without accidentally overwriting an update that was done simultaneously.

content - The "content" is a JSON encrypted message that includes actual input for the SD update. The standard JSON content encryption key (CEK) is used, and the CEK is encrypted by the target TEE's public key.

tamid - SD owner claim by TAM - An SD owned by a TAM will be associated with a trusted identifier defined as an attribute in the signer TAM certificate.

spid - SP identifier of the TA owner SP

sdname - the name of the target SD where the TA is to be installed

spcert - an optional field to specify the SP certificate that signed the TA. This is sent if the SP has a new certificate that hasn't been previously registered with the target SD where the TA should be installed.

taid - the identifier of the TA application to be installed

encrypted_ta - the message portion contains encrypted TA binary data and personalization data. The TA data encryption key is placed in "key", which is encrypted by the recipient's public key, using JWE enveloped structure. The TA data encryption uses symmetric key based encryption such as AESCBC.

According to the OTrP message template, the full request InstallTAResponse is a signed message over the InstallTATBSRequest as follows.

```
{
  "InstallTAResponse": {
    "payload": "<InstallTATBSRequest JSON above>",
    "protected": "<integrity-protected header contents>",
    "header": "<non-integrity-protected header contents>",
    "signature": "<signature contents signed by TAM private key>"
  }
}
```

9.3.1.2. InstallTAResponse Message

The response message for a InstallTAResponse contains the following content.

```

{
  "InstallTATBSResponse": {
    "ver": "1.0",
    "status": "<operation result>",
    "rid": "<the request ID received>",
    "tid": "<the transaction ID received>",
    "content": ENCRYPTED {
      "reason": "<failure reason detail>", // optional
      "did": "<the device id hash>",
      "dsi": "<Updated TEE state, including all SD owned by
              this TAM>"
    }
  }
}

```

In the response message, the following fields MUST be supplied.

did - the SHA256 hash of the device TEE certificate. This shows the device ID explicitly to the receiving TAM.

The final message InstallTAResponse looks like the following.

```

{
  "InstallTAResponse": {
    "payload": "<InstallTATBSResponse JSON above>",
    "protected": {
      "<BASE64URL of signing algorithm>"
    },
    "signature": "<signature contents signed by TEE device
                  private key (BASE64URL)>"
  }
}

```

When the TEE fails to respond, the OTrP Broker will not provide a subsequent response to the TAM. The TAM should treat this as if the device has gone offline where a response is never delivered back.

9.3.1.3. Error Conditions

An error may occur if a request isn't valid or the TEE runs into some error. The list of possible errors are as follows. Refer to the Error Code List (Section 13.1) for detailed causes and actions.

ERR_REQUEST_INVALID

ERR_UNSUPPORTED_MSG_VERSION

ERR_UNSUPPORTED_CRYPTO_ALG

ERR_DEV_STATE_MISMATCH

ERR_SD_NOT_FOUND

ERR_TA_INVALID

ERR_TA_ALREADY_INSTALLED

ERR_TEE_FAIL

ERR_TEE_RESOURCE_FULL

ERR_TAM_NOT_AUTHORIZED

ERR_TAM_NOT_TRUSTED

9.3.2. UpdateTA

This TAM-initiated command can update a TA and its data in an SP's SD that it manages for the following purposes.

1. Update TA binary
2. Update TA's personalization data

The TAM presents the proof of the SD ownership to a TEE, and includes related information in its signed message. The entire request is also encrypted for end-to-end confidentiality.

The TEE processes the command from the TAM to update the TA of an SP SD. It does the following:

- o Validation
 - * The TEE validates the TAM message authenticity
 - * Decrypt to get request content
 - * Look up the SD with the SD name
 - * Checks that the TAM owns the SD
 - * Checks DSI hash matches that the device state hasn't changed
- o TA validation

- * Both TA binary and personalization data are optional, but at least one of them shall be present in the message
- * Decrypt to get the TA binary and any personalization data with the "TEE SP AIK private key"
- * Check that SP ID is the one that is registered with the SP SD
- * Check that the TA signer is either a newly given SP certificate or the one in SD.
- * If a TA signer is given in the request, add this signer into the SD.
- o If the above validation passes, continue to do TA update
 - * The TEE re-encrypts the TA binary and its personalization data with its own method
 - * The TEE replaces the existing TA binary and its personalization data with the new binary and data.
- o Construct a response message. This involves signing a encrypted status information for the requesting TAM.

9.3.2.1. UpdateTAResponse Message

The request message for UpdateTA has the following JSON format.

```
{
  "UpdateTATBSRequest": {
    "ver": "1.0",
    "rid": "<unique request ID>",
    "tid": "<transaction ID>",
    "tee": "<TEE routing name from the DSI for the SD's target>",
    "nextdsi": true | false,
    "dsihash": "<hash of DSI returned in the prior query>",
    "content": ENCRYPTED {
      "tamid": "<TAM ID previously assigned to the SD>",
      "spid": "<SPID value>",
      "sdname": "<SD name for the domain to be created>",
      "spcert": "<BASE64 encoded SP certificate >", // optional
      "taid": "<TA identifier>"
    },
    "encrypted_ta": {
      "key": "<JWE enveloped data of a 256-bit symmetric key by
        the recipient's TEEspaik public key>",
      "iv": "<hex of 16 random bytes>",
      "alg": "<encryption algorithm. AESCBC by default.>",
      "ciphernewtadata": "<Change existing TA binary to this new TA
        binary data(BASE64 encoded and encrypted)>",
      "ciphernewpdata": "<Change the existing data to this new TA
        personalization data(BASE64 encoded and encrypted)>"
      // optional
    }
  }
}
```

In the message,

rid - A unique value to identify this request

tid - A unique value to identify this transaction. It can have the same value for the tid in the preceding GetDeviceStateRequest.

tee - TEE ID returned from the previous GetDeviceStateResponse

nextdsi - Indicates whether the up-to-date Device State Information (DSI) is to be returned in the response to this request.

dsihash - The BASE64-encoded SHA256 hash value of the DSI data returned in the prior TAM operation with this target TEE. This value is always included such that a receiving TEE can check whether the device state has changed since its last query. It

helps enforce SD update order in the right sequence without accidentally overwriting an update that was done simultaneously.

content - The "content" is a JSON encrypted message that includes actual input for the SD update. The standard JSON content encryption key (CEK) is used, and the CEK is encrypted by the target TEE's public key.

tamid - SD owner claim by TAM - an SD owned by a TAM will be associated with a trusted identifier defined as an attribute in the signer TAM certificate.

spid - SP identifier of the TA owner SP

spcert - an optional field to specify the SP certificate that signed the TA. This is sent if the SP has a new certificate that hasn't been previously registered with the target SD where the TA is to be installed.

sdname - the name of the target SD where the TA should be updated

taid - an identifier for the TA application to be updated

encrypted_ta - the message portion contains newly encrypted TA binary data and personalization data.

According to the OTrP message template, the full request UpdateTARequest is a signed message over the UpdateTATBSRequest as follows.

```
{
  "UpdateTARequest": {
    "payload": "<UpdateTATBSRequest JSON above>",
    "protected": "<integrity-protected header contents>",
    "header": "<non-integrity-protected header contents>",
    "signature": "<signature contents signed by TAM private key>"
  }
}
```

9.3.2.2. UpdateTAResponse Message

The response message for a UpdateTARequest contains the following content.

```

{
  "UpdateTATBSResponse": {
    "ver": "1.0",
    "status": "<operation result>",
    "rid": "<the request ID received>",
    "tid": "<the transaction ID received>",
    "content": ENCRYPTED {
      "reason": "<failure reason detail>", // optional
      "did": "<the device id hash>",
      "dsi": "<Updated TEE state, including all SD owned by
             this TAM>"
    }
  }
}

```

In the response message, the following fields MUST be supplied.

did - the SHA256 hash of the device TEE certificate. This shows the device ID explicitly to the receiving TAM.

The final message UpdateTAResponse looks like the following.

```

{
  "UpdateTAResponse": {
    "payload": "<UpdateTATBSResponse JSON above>",
    "protected": {
      "<BASE64URL of signing algorithm>"
    },
    "signature": "<signature contents signed by TEE device
                 private key (BASE64URL)>"
  }
}

```

When the TEE fails to respond, the OTrP Broker will not provide a subsequent response to the TAM. The TAM should treat this as if the device has gone offline where a response is never delivered back.

9.3.2.3. Error Conditions

An error may occur if a request isn't valid or the TEE runs into some error. The list of possible errors are as follows. Refer to the Error Code List (Section 13.1) for detailed causes and actions.

ERR_REQUEST_INVALID

ERR_UNSUPPORTED_MSG_VERSION

ERR_UNSUPPORTED_CRYPTO_ALG

ERR_DEV_STATE_MISMATCH

ERR_SD_NOT_FOUND

ERR_TA_INVALID

ERR_TA_NOT_FOUND

ERR_TEE_FAIL

ERR_TAM_NOT_AUTHORIZED

ERR_TAM_NOT_TRUSTED

9.3.3. DeleteTA

This operation defines OTrP messages that allow a TAM to instruct a TEE to delete a TA for an SP in a given SD. A TEE will delete a TA from an SD and also TA data in the TEE. A Client Application cannot directly access TEE or OTrP Broker to delete a TA.

9.3.3.1. DeleteTATBSRequest Message

The request message for DeleteTA has the following JSON format.

```
{
  "DeleteTATBSRequest": {
    "ver": "1.0",
    "rid": "<unique request ID>",
    "tid": "<transaction ID>",
    "tee": "<TEE routing name from the DSI for the SD's target>",
    "nextdsi": true | false,
    "dsihash": "<hash of DSI returned in the prior query>",
    "content": ENCRYPTED {
      "tamid": "<TAM ID previously assigned to the SD>",
      "sdname": "<SD name of the TA>",
      "taid": "<the identifier of the TA to be deleted from the
              specified SD>"
    }
  }
}
```

In the message,

rid - A unique value to identify this request

tid - A unique value to identify this transaction. It can have the same value for the tid in the preceding GetDeviceStateRequest.

tee - The TEE ID returned from the previous GetDeviceStateResponse

nextdsi - Indicates whether the up-to-date Device State Information (DSI) is to be returned in the response to this request.

dsihash - The BASE64-encoded SHA256 hash value of the DSI data returned in the prior TAM operation with this target TEE. This value is always included such that a receiving TEE can check whether the device state has changed since its last query. It helps enforce SD update order in the right sequence without accidentally overwriting an update that was done simultaneously.

content - The "content" is a JSON encrypted message that includes actual input for the SD update. The standard JSON content encryption key (CEK) is used, and the CEK is encrypted by the target TEE's public key.

tamid - SD owner claim by TAM - an SD owned by a TAM will be associated with a trusted identifier defined as an attribute in the signer TAM certificate.

sdname - the name of the target SD where the TA is installed

taid - an identifier for the TA application to be deleted

According to the OTrP message template, the full request DeleteTAResponse is a signed message over the DeleteTATBSRequest as follows.

```
{
  "DeleteTAResponse": {
    "payload": "<DeleteTATBSRequest JSON above>",
    "protected": "<integrity-protected header contents>",
    "header": "<non-integrity-protected header contents>",
    "signature": "<signature contents signed by TAM
                  private key>"
  }
}
```

9.3.3.2. Request Processing Requirements at a TEE

A TEE processes a command from a TAM to delete a TA of an SP SD. It does the following:

1. Validate the JSON request message
 - * The TEE validates TAM message authenticity
 - * Decrypt to get request content
 - * Look up the SD and the TA with the given SD name and TA ID
 - * Checks that the TAM owns the SD, and TA is installed in the SD
 - * Checks that the DSI hash matches and the the device state hasn't changed
2. Deletion action
 - * If all the above validation points pass, the TEE deletes the TA from the SD
 - * The TEE SHOULD also delete all personalization data for the TA
3. Construct DeleteTAResponse message.

If a request is illegitimate or the signature doesn't pass, a "status" property in the response will indicate the error code and cause.

9.3.3.3. DeleteTAResponse Message

The response message for a DeleteTARequest contains the following content.

```

{
  "DeleteTATBSResponse": {
    "ver": "1.0",
    "status": "<operation result>",
    "rid": "<the request ID received>",
    "tid": "<the transaction ID received>",
    "content": ENCRYPTED {
      "reason": "<failure reason detail>", // optional
      "did": "<the device id hash>",
      "dsi": "<Updated TEE state, including all SD owned by
             this TAM>"
    }
  }
}

```

In the response message, the following fields MUST be supplied.

did - the SHA256 hash of the device TEE certificate. This shows the device ID explicitly to the receiving TAM.

The final message DeleteTAResponse looks like the following.

```

{
  "DeleteTAResponse": {
    "payload": "<DeleteTATBSResponse JSON above>",
    "protected": {
      "<BASE64URL of signing algorithm>"
    },
    "signature": "<signature contents signed by TEE device
                 private key (BASE64URL)>"
  }
}

```

When the TEE fails to respond, the OTrP Broker will not provide a subsequent response to the TAM. The TAM should treat this as if the device has gone offline where a response is never delivered back.

9.3.3.4. Error Conditions

An error may occur if a request isn't valid or the TEE runs into some error. The list of possible errors are as follows. Refer to the Error Code List (Section 13.1) for detailed causes and actions.

ERR_REQUEST_INVALID

ERR_UNSUPPORTED_MSG_VERSION

ERR_UNSUPPORTED_CRYPTO_ALG

ERR_DEV_STATE_MISMATCH

ERR_SD_NOT_FOUND

ERR_TA_NOT_FOUND

ERR_TEE_FAIL

ERR_TAM_NOT_AUTHORIZED

ERR_TAM_NOT_TRUSTED

10. Response Messages a TAM May Expect

A TAM expects some feedback from a remote device when a request message is delivered to a device. The following three types of responses SHOULD be supplied.

Type 1: Expect a valid TEE-generated response message

A valid TEE signed response may contain errors detected by a TEE, e.g. a TAM is trusted but some TAM-supplied data is missing, for example, SP ID doesn't exist. TEE MUST be able to sign and encrypt.

If a TEE isn't able to sign a response, the TEE returns an error to the OTrP Broker without giving any other internal information. The OTrP Broker will be generating the response.

Type 2: The OTrP Broker generated error message when TEE fails. OTrP Broker errors will be defined in this document.

A Type 2 message has the following format.

```
{
  "OTrPBrokerError": {
    "ver": "1.0",
    "rid": "",
    "tid": "",
    "errcode": "ERR_AGENT_TEE_UNKNOWN | ERR_AGENT_TEE_BUSY"
  }
}
```

Type 3: OTrP Broker itself isn't reachable or fails. A Client Application is responsible to handle error and respond the TAM in its own way. This is out of scope for this specification.

11. Basic Protocol Profile

This section describes a baseline for interoperability among the protocol entities, mainly, the TAM and TEE.

A TEE MUST support RSA algorithms. It is optional to support ECC algorithms. A TAM SHOULD use a RSA certificate for TAM message signing. It may use an ECC certificate if it detects that the TEE supports ECC according to the field "supportedsigalgs" in a TEE response.

A TAM MUST support both RSA 2048-bit algorithm and ECC P-256 algorithms. With this, a TEE and TFW certificate can be either RSA or ECC type.

JSON signing algorithms

- o RSA PKCS#1 with SHA256 signing : "RS256"
- o ECDSA with SHA256 signing : "ES256"

JSON asymmetric encryption algorithms (describes key-exchange or key-agreement algorithm for sharing symmetric key with TEE):

- o RSA PKCS#1 : "RSA1_5"
- o ECDH using TEE ECC P-256 key and ephemeral ECC key generated by TAM : "ECDH-ES+A128W"

JSON symmetric encryption algorithms (describes symmetric algorithm for encrypting body of data, using symmetric key transferred to TEE using asymmetric encryption):

- o Authenticated encryption AES 128 CBC with SHA256 :
{"enc": "A128CBC-HS256"}

12. Attestation Implementation Consideration

It is important to know that the state of a device is appropriate before trusting that a device is what it says it is. The attestation scheme for OTrP must also be able to cope with different TEEs, including those that are OTrP compliant and those that use another mechanism. In the initial version, only one active TEE is assumed.

It is out of scope how the TAM and the device implement the trust hierarchy verification. However, it is helpful to understand what each system provider should do in order to properly implement an OTrP trust hierarchy.

In this section, we provide some implementation reference consideration.

12.1. OTrP Trusted Firmware

12.1.1. Attestation signer

It is proposed that attestation for OTrP is based on the TFW layer, and that further attestation is not performed within the TEE itself during Security Domain operations. The rationale is that the device boot process will be defined to start with a secure bootloader protected with a harden key in eFUSE. The process releases attestation signing capabilities into the TFW once a trust boot has been established. In this way the release of the attestation signer can be considered the first "platform configuration metric", using Trust Computing Group (TCG) terminology.

12.1.2. TFW Initial Requirements

- R1 The TFW must be possible for verification during boot
- R2 The TFW must allow a public / private key pair to be generated during device manufacture
- R3 The public key and certificate must be possible to store securely
- R4 The private key must be possible to store encrypted at rest
- R5 The private key must only be visible to the TFW when it is decrypted
- R6 The TFW must be able to read a list of root and intermediate certificates that it can use to check certificate chains with. The list must be stored such that it cannot be tampered with
- R7 Need to allow a TEE to access its unique TEE specific private key

12.2. TEE Loading

During boot, the TFW is required to start all of the root TEEs. Before loading them, the TFW must first determine whether the code sign signature of the TEE is valid. If TEE integrity is confirmed, the TEE may be started. The TFW must then be able to receive the identity certificate from the TEE (if that TEE is OTrP compliant). The identity certificate and keys will need to be baked into the TEE image, and therefore also covered by the code signer hash during the manufacturing process. The private key for the identity certificate must be securely protected. The private key for a TEE identity must

never be released no matter how the public key and certificate are released to the TFW.

Once the TFW has successfully booted a TEE and retrieved the identity certificate, the TFW will commit this to the platform configuration register (PCR) set, for later use during attestation. At minimum, the following data must be committed to the PCR for each TEE:

1. Public key and certificate for the TEE
2. TEE identifier that can be used later by a TAM to identify this TEE

12.3. Attestation Hierarchy

The attestation hierarchy and seed required for TAM protocol operation must be built into the device at manufacture. Additional TEEs can be added post-manufacture using the scheme proposed, but it is outside of the current scope of this document to detail that.

It should be noted that the attestation scheme described is based on signatures. The only decryption that may take place is through the use of a bootloader key.

12.3.1. Attestation Hierarchy Establishment: Manufacture

During manufacture the following steps are required:

1. A device-specific TFW key pair and certificate are burnt into the device. This key pair will be used for signing operations performed by the TFW.
2. TEE images are loaded and include a TEE instance-specific key pair and certificate. The key pair and certificate are included in the image and covered by the code signing hash.
3. The process for TEE images is repeated for any subordinate TEEs, which are additional TEEs after the root TEE that some devices have.

12.3.2. Attestation Hierarchy Establishment: Device Boot

During device boot the following steps are required:

1. The boot module releases the TFW private key by decrypting it with the bootloader key.

2. The TFW verifies the code-signing signature of the active TEE and places its TEE public key into a signing buffer, along with its identifier for later access. For a non-OTrP TEE, the TFW leaves the TEE public key field blank.
3. The TFW signs the signing buffer with the TFW private key.
4. Each active TEE performs the same operation as the TFW, building up their own signed buffer containing subordinate TEE information.

12.3.3. Attestation Hierarchy Establishment: TAM

Before a TAM can begin operation in the marketplace to support devices of a given TEE, it must obtain a TAM certificate from a CA that is registered in the trust store of devices with that TEE. In this way, the TEE can check the intermediate and root CA and verify that it trusts this TAM to perform operations on the TEE.

13. IANA Considerations

There are two IANA requests: a media type and list of error codes.

This section first requests that IANA assign a media type: application/otrp+json.

Type name: application

Subtype name: otp+json

Required parameters: none

Optional parameters: none

Encoding considerations: Same as encoding considerations of application/json as specified in Section 11 of [RFC7159]

Security considerations: See Section 12 of [RFC7159] and Section 14 of this document

Interoperability considerations: Same as interoperability considerations of application/json as specified in [RFC7159]

Published specification: [TEEPArch]

Applications that use this media type: OTrP 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

Person to contact for further information: teep@ietf.org

Intended usage: COMMON

Restrictions on usage: none

Author: See the "Authors' Addresses" section of this document

Change controller: IETF

The error code listed in the next section will be registered.

13.1. Error Code List

This section lists error codes that could be reported by a TA or TEE in a device in responding to a TAM request, and a separate list that OTrP Broker may return when the TEE fails to respond.

13.1.1. TEE Signed Error Code List

ERR_DEV_STATE_MISMATCH - A TEE will return this error code if the DSI hash value from TAM doesn't match the has value of the device's current DSI.

ERR_SD_ALREADY_EXISTS - This error will occur if an SD to be created already exists in the TEE.

ERR_SD_NOT_EMPTY - This is reported if a target SD isn't empty.

ERR_SDNAME_ALREADY_USED A TEE will return this error code if the new SD name already exists in the TEE.

ERR_REQUEST_INVALID - This error will occur if the TEE meets any of the following conditions with a request message: (1) The request from a TAM has an invalid message structure; mandatory information is absent in the message. undefined member or structure is included. (2) TEE fails to verify signature of the message or fails to decrypt its contents.

ERR_SPCERT_INVALID - If a new SP certificate for the SD to be updated is not valid, then the TEE will return this error code.

ERR_TA_ALREADY_INSTALLED - While installing a TA, a TEE will return this error if the TA has already been installed in the SD.

ERR_TA_INVALID - This error will occur when a TEE meets any of following conditions while checking validity of TA: (1) The TA binary has a format that the TEE can't recognize. (2) The TEE fails to decrypt the encoding of the TA binary and personalization data. (3) If an SP isn't registered with the SP SD where the TA will be installed.

ERR_TA_NOT_FOUND - This error will occur when the target TA doesn't exist in the SD.

ERR_TEE_FAIL - If the TEE fails to process a request because of an internal error, it will return this error code.

ERR_TEE_RESOURCE_FULL - This error is reported when a device resource isn't available anymore such as storage space is full.

ERR_TFW_NOT_TRUSTED - A TEE is responsible for determining that the underlying device firmware is trustworthy. If the TEE determines the TFW is not trustworthy, then this error will occur.

ERR_TAM_NOT_TRUSTED - Before processing a request, a TEE needs to make sure whether the sender TAM is trustworthy by checking the validity of the TAM certificate, etc. If the TEE finds that the TAM is not trustworthy, then it will return this error code.

ERR_UNSUPPORTED_CRYPTO_ALG - This error will occur if a TEE receives a request message encoded with cryptographic algorithms that the TEE doesn't support.

ERR_UNSUPPORTED_MSG_VERSION - This error will occur if a TEE receives a message version that the TEE can't deal with.

14. Security Consideration

14.1. Cryptographic Strength

The strength of the cryptographic algorithms, using the measure of 'bits of security' defined in NIST SP800-57 allowed for OTrP is:

- o At a minimum, 112 bits of security. The limiting factor for this is the RSA-2048 algorithm, which is indicated as providing 112 bits of symmetric key strength in SP800-57. It is important that

RSA is supported in order to enhance the interoperability of the protocol.

- o The option exists to choose algorithms providing 128 bits of security. This requires using TEE devices that support ECC P256.

The available algorithms and key sizes specified in this document are based on industry standards. Over time the recommended or allowed cryptographic algorithms may change. It is important that the OTrP allows for crypto-agility. In this specification, TAM and TEE can negotiate an agreed upon algorithm where both include their supported algorithm in OTrP message.

14.2. Message Security

OTrP messages between the TAM and TEE are protected by message security using JWS and JWE. The 'Basic protocol profile' section of this document describes the algorithms used for this. All OTrP TEE devices and OTrP TAMs must meet the requirements of the basic profile. In the future additional 'profiles' can be added.

PKI is used to ensure that the TEE will only communicate with a trusted TAM, and to ensure that the TAM will only communicate with a trusted TEE.

14.3. TEE Attestation

It is important that the TAM can trust that it is talking to a trusted TEE. This is achieved through attestation. The TEE has a private key and certificate built into it at manufacture, which is used to sign data supplied by the TAM. This allows the TAM to verify that the TEE is trusted.

It is also important that the TFW (trusted firmware) can be checked. The TFW has a private key and certificate built into it at manufacture, which allows the TEE to check that that the TFW is trusted.

The GetDeviceState message therefore allows the TAM to check that it trusts the TEE, and the TEE at this point will check whether it trusts the TFW.

14.4. TA Protection

A TA will be delivered in an encrypted form. This encryption is an additional layer within the message encryption described in the Section 11 of this document. The TA binary is encrypted for each target device with the device's TEE SP AIK public key. A TAM can

either do this encryption itself or provide the TEE SP AIK public key to an SP such that the SP encrypts the encrypted TA for distribution to the TEE.

The encryption algorithm can use a random AES 256 key "taek" with a 16 byte random IV, and the "taek" is encrypted by the "TEE SP AIK public key". The following encrypted TA data structure is expected by a TEE:

```
"encrypted_ta_bin": {
  "key": "<JWE enveloped data of a 256-bit symmetric key by
        the recipient's TEEspaik public key>",
  "iv": "<hex of 16 random bytes>",
  "alg": "AESCBC",
  "cipherdata": "<BASE64 encoded encrypted TA binary data>"
}
```

14.5. TA Personalization Data

An SP or TAM can supply personalization data for a TA to initialize for a device. Such data is passed through an InstallTA command from a TAM. The personalization data itself is (or can be) opaque to the TAM. The data can be from the SP without being revealed to the TAM. The data is sent in an encrypted manner in a request to a device such that only the device can decrypt. A device's TEE SP AIK public key for an SP is used to encrypt the data. Here JWE enveloping is used to carry all encryption key parameters along with encrypted data.

```
"encrypted_ta_data": { // "TA personalization data"
  "key": "<JWE enveloped data of a 256-bit symmetric key by
        the recipient's TEEspaik public key>",
  "iv": "<hex of 16 random bytes>",
  "alg": "AESCBC",
  "cipherdata": "<BASE64 encoded encrypted TA personalization
                data>"
}
```

14.6. TA Trust Check at TEE

A TA binary is signed by a TA signer certificate. This TA signing certificate/private key belongs to the SP, and may be self-signed (i.e., it need not participate in a trust hierarchy). It is the responsibility of the TAM to only allow verified TAs from trusted SPs into the system. Delivery of that TA to the TEE is then the responsibility of the TEE, using the security mechanisms provided by the OTrP.

We allow a way for an (untrusted) application to check the trustworthiness of a TA. OTrP Broker has a function to allow a Client Application to query the information about a TA.

An application in the Rich O/S may perform verification of the TA by verifying the signature of the TA. The GetTAInformation function is available to return the TEE supplied TA signer and TAM signer information to the application. An application can do additional trust checks on the certificate returned for this TA. It might trust the TAM, or require additional SP signer trust chaining.

14.7. One TA Multiple SP Case

A TA for multiple SPs must have a different identifier per SP. A TA will be installed in a different SD for each respective SP.

14.8. OTrP Broker Trust Model

An OTrP Broker could be malware in the vulnerable REE. A Client Application will connect its TAM provider for required TA installation. It gets command messages from the TAM, and passes the message to the OTrP Broker.

The OTrP is a conduit for enabling the TAM to communicate with the device's TEE to manage SDs and TAs. All TAM messages are signed and sensitive data is encrypted such that the OTrP Broker cannot modify or capture sensitive data.

14.9. OCSF Stapling Data for TAM Signed Messages

The GetDeviceStateRequest message from a TAM to a TEE shall include OCSF stapling data for the TAM's signer certificate and for intermediate CA certificates up to the root certificate so that the TEE can verify the signer certificate's revocation status.

A certificate revocation status check on a TA signer certificate is OPTIONAL by a TEE. A TAM is responsible for vetting a TA and the SP before it distributes them to devices. A TEE will trust a TA signer certificate's validation status done by a TAM when it trusts the TAM.

14.10. Data Protection at TAM and TEE

The TEE implementation provides protection of data on the device. It is the responsibility of the TAM to protect data on its servers.

14.11. Privacy Consideration

Devices are issued with a unique TEE certificate to attest the device's validity. This uniqueness also creates a privacy and tracking risk that must be mitigated.

The TEE will only release the TEE certificate to a trusted TAM (it must verify the TAM certificate before proceeding). OTrP is designed such that only a TAM can obtain the TEE device certificate and firmware certificate - the GetDeviceState message requires signature checks to validate the TAM is trusted, and OTrP delivers the device's certificate(s) encrypted such that only that TAM can decrypt the response. A Client Application will never see the device certificate.

An SP-specific TEE SP AIK (TEE SP Anonymous Key) is generated by the protocol for Client Applications. This provides a way for the Client Application to validate some data that the TEE may send without requiring the TEE device certificate to be released to the client device rich O/S, and to optionally allow an SP to encrypt a TA for a target device without the SP needing to be supplied with the TEE device certificate.

14.12. Threat Mitigation

A rogue application may perform excessive TA loading. An OTrP Broker implementation should protect against excessive calls.

Rogue applications might request excessive SD creation. The TAM is responsible to ensure this is properly guarded against.

Rogue OTrP Broker could replay or send TAM messages out of sequence: e.g., a TAM sends update1 and update2. The OTrP Broker replays update2 and update1 again, creating an unexpected result that a client wants. "dsihash" is used to mitigate this. The TEE MUST store DSI state and check that the DSI state matches before it does another update.

Concurrent calls from a TAM to a TEE MUST be handled properly by a TEE. If multiple concurrent TAM operations take place, these could fail due to the "dsihash" being modified by another concurrent operation. The TEE is responsible for resolve any locking such that one application cannot lock other applications from using the TEE, except for a short term duration of the TAM operation taking place. For example, an OTrP operation that starts but never completes (e.g. loss of connectivity) must not prevent subsequent OTrP messages from being executed.

14.13. Compromised CA

A root CA for TAM certificates might get compromised. Some TEE trust anchor update mechanism is expected from device OEMs. A compromised intermediate CA is covered by OCSP stapling and OCSP validation check in the protocol. A TEE should validate certificate revocation about a TAM certificate chain.

If the root CA of some TEE device certificates is compromised, these devices might be rejected by a TAM, which is a decision of the TAM implementation and policy choice. Any intermediate CA for TEE device certificates SHOULD be validated by TAM with a Certificate Revocation List (CRL) or Online Certificate Status Protocol (OCSP) method.

14.14. Compromised TAM

The TEE SHOULD use validation of the supplied TAM certificates and OCSP stapled data to validate that the TAM is trustworthy.

Since PKI is used, the integrity of the clock within the TEE determines the ability of the TEE to reject an expired TAM certificate, or revoked TAM certificate. Since OCSP stapling includes signature generation time, certificate validity dates are compared to the current time.

14.15. Certificate Renewal

TFW and 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 update. On the other hand, it is imperative that OEMs or device providers plan for support of trust anchor update in their shipped devices.

15. Acknowledgements

We thank Alin Mutu for his contribution to many discussion that helped to design the trust flow mechanisms, and the creation of the flow diagrams. We also thank the following people (in alphabetical order) for their input and review: Sangsu Baek, Rob Coombs, Dapeng Liu, Dave Thaler, and Pengfei Zhao.

16. References

16.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.
- [RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<https://www.rfc-editor.org/info/rfc7516>>.
- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<https://www.rfc-editor.org/info/rfc7517>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<https://www.rfc-editor.org/info/rfc7518>>.
- [TEEPArch] Pei, M., Tschofenig, H., Atyeo, A., and D. Liu, "Trusted Execution Environment Provisioning (TEEP) Architecture", 2018, <<https://tools.ietf.org/html/draft-ietf-teep-architecture-02>>.

16.2. Informative References

- [GPTEE] Global Platform, "Global Platform, GlobalPlatform Device Technology: TEE System Architecture, v1.0", 2013.
- [GPTEECLAPI] Global Platform, "Global Platform, GlobalPlatform Device Technology: TEE Client API Specification, v1.0", 2013.

Appendix A. Sample Messages

A.1. Sample Security Domain Management Messages

A.1.1. Sample GetDeviceState

A.1.1.1. Sample GetDeviceStateRequest

The TAM builds a "GetDeviceStateTBSRequest" message.

```
{
  "GetDeviceStateTBSRequest": {
    "ver": "1.0",
    "rid": "8C6F9DBB-FC39-435c-BC89-4D3614DA2F0B",
    "tid": "4F454A7F-002D-4157-884E-B0DD1A06A8AE",
    "ocspdat": "c2FtcGx1IG9jc3BkYXQgQjY0IGVuY29kZWQgQVNOMQ==",
    "icaocspdat": "c2FtcGx1IGl jYW9jc3BkYXQgQjY0IGVuY29kZWQgQVNOMQ==",
    "supportedsigalgs": "RS256"
  }
}
```

The TAM signs "GetDeviceStateTBSRequest", creating "GetDeviceStateRequest"

```
{
  "GetDeviceStateRequest": {
    "payload": "
ewoJlkdldERldmldjZVN0YXRlVEJTUmVxdWVzdCI6IHsKCQkidmVyIjogIjEuMCIsCgkJ
InJpZCI6IHs4QzZGOURCQilGQzM5LTQzNWmtQkM4OS00RDM2MTREQTJGMEJ9LAoJCSJ0
aWQiOiAieZRGNDU0QTdGLTAwMkQtNDElNy04ODRFLUIwREQxQTA2QThBRX0iLAoJCSJv
Y3NwZGF0IjogImMyRnRjR3hsSUc5amMzQmtZWFFnUWpZME1HVnVZMjlrWldRZ1FWTk9N
UT09IiwKCQkiaWNhb2NzcGRhdCI6ICJjMkZ0Y0d4bElHbGpZVz1qYzNCa1lYUWdRalkw
SUDWdVkyOWtaV1FnUVZOT01RPT0iLAoJCSJzdXBwb3J0ZWRzaWdhbGdzIjogIlJTMjU2
IgoJfQp9",
    "protected": "eyJhbGciOiJSUzI1NiJ9",
    "header": {
      "x5c": ["ZXhhbXBsZSBBU04xIHNPZ251ciBjZXJ0aWZpY2F0ZQ==",
        "ZXhhbXBsZSBBU04xIENBIGNlcnRpZmljYXRl"]
    },
    "signature": "c2FtcGx1IHNPZ25hdHVyZQ"
  }
}
```

A.1.1.2. Sample GetDeviceStateResponse

The TAM sends "GetDeviceStateRequest" to the OTrP Broker

The OTrP Broker obtains "dsi" from each TEE. (In this example there is a single TEE.)

The TEE obtains signed "fwdata" from firmware.

The TEE builds "dsi" - summarizing device state of the TEE.

```

{
  "dsi": {
    "tfwdata": {
      "tbs": "ezRGNDU0QTdGLTAwMkQtNDE1Ny04ODRFLUIwREQxQTA2QThBRX0=",
      "cert": "ZXhhbXBsZSBGVyBjZXJ0aWZpY2F0ZQ==",
      "sigalg": "RS256",
      "sig": "c2FtcGx1IEZXIHNPZ25hdHVyZQ=="
    },
    "tee": {
      "name": "Primary TEE",
      "ver": "1.0",
      "cert": "c2FtcGx1IFRFRSBjZXJ0aWZpY2F0ZQ==",
      "cacert": [
        "c2FtcGx1IENBIGNlcnRpZmljYXRlIDE=",
        "c2FtcGx1IENBIGNlcnRpZmljYXRlIDI="
      ],
      "sdlist": {
        "cnt": "1",
        "sd": [
          {
            "name": "default.acmebank.com",
            "spid": "acmebank.com",
            "talist": [
              {
                "taid": "acmebank.secure.banking",
                "taname": "Acme secure banking app"
              },
              {
                "taid": "acmebank.loyalty.rewards",
                "taname": "Acme loyalty rewards app"
              }
            ]
          }
        ]
      },
      "teeaiklist": [
        {
          "spaik": "c2FtcGx1IEFTTjEgZW5jb2RlZCBQS0NTMSBwdWJsaWNrZXk=",
          "spaiktype": "RSA",
          "spid": "acmebank.com"
        }
      ]
    }
  }
}

```

The TEE encrypts "dsi", and embeds it into a "GetDeviceTEEStateTBSResponse" message.

```

{
  "GetDeviceTEEStateTBSResponse": {
    "ver": "1.0",
    "status": "pass",
    "rid": "{8C6F9DBB-FC39-435c-BC89-4D3614DA2F0B}",
    "tid": "{4F454A7F-002D-4157-884E-B0DD1A06A8AE}",
    "signerreq": "false",
    "edsi": {
      "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0K",
      "recipients": [
        {
          "header": {
            "alg": "RSA1_5"
          },
          "encrypted_key":
            "QUVTMTI4IChDRUspIGtleSwgZW5jcnlwdGVkIHdpdGggVFNNIFJTQSBwdWJsaWMga2V5LCB1c2luZyBSU0ExXzUgcGFkZGluZW"
        }
      ],
      "iv": "ySGmfZ69YlcEilNr5_SGbA",
      "ciphertext":
        "c2FtcGx1IGRzaSBkYXRhIGVuY3J5cHRlZCB3aXRoIEFFUzEyOCBrZXkgZnJvbSB5ZW50cGllbnRzLmVuY3J5cHRlZF9rZXk",
      "tag": "c2FtcGx1IGF1dGhlbnRpY2F0aW9uIHRhZw"
    }
  }
}

```

The TEE signs "GetDeviceTEEStateTBSResponse" and returns it to the OTrP Broker. The OTrP Broker encodes "GetDeviceTEEStateResponse" into an array to form "GetDeviceStateResponse".

```

{
  "GetDeviceStateResponse": [
    {
      "GetDeviceTEEStateResponse": {
        "payload":
          "ewogICJHZXREZXZpY2VURUVTdGF0ZVRVCU1Jlc3Bvb3N1IjogewogICAgInZlciI6
          ICixLjAiLAogICAgInN0YXR1cyI6ICJwYXNzIiwKICAgICJyaWQiOiAiezhDNkY5
          REJCLUZDMzktNDM1Yy1CQzg5LTREMzYxNERBMkYwQn0iLAogICAgInRpZCI6ICJ7
          NEY0NTRBN0YtMDAyRC00MTU3LTg4NEUtQjBERDFBMDZBOEFFfSIsCgkic2lnbmVy
          cmVxIjoizMfsc2UiLAogICAgImVkc2kiOiB7CiAgICAgICJwcm90ZWN0ZWQiOiAi
          ZX1KbGJtTW1PaUpCTVRJNFwSkRMVWhUTWpVMkluMESiLAogICAgICAicmVjaXBp
          ZW50cyI6IFsKICAgICAgICB7CiAgICAgICAgICAiaGVhZGVyIjoigewogICAgICAg
          ICAgImFsZyI6ICJSU0ExXzUiCiAgICAgICAgfSwKICAgICAgICAiZW5jcnldGVk
          X2tleSI6CiAgICAgICAgIogogICAgICAgIFFVVLjRNVEk0SUNoRFJvc3BJR3RsZVN3
          Z1pXNWpjbmx3ZEdWa01lZHBkR2dnVkdZOTk1GS1RRU0J3ZFdkc2FXTWcKICAgICAg
          ICBhM1Y1TENCMWMybHVaeUJTVTBFeFh6VWdjR0ZrWkdsdVp3IogogICAgICAgIH0K
          ICAgICAgXSwwKICAgICAgIml2IjoigInlTR2lmWjY5WWxjRWlsTnI1X1NHYkEiLAog
          ICAgICAiY2lwaGVydGV4dCI6CiAgICAgICIKICAgICAgYzJGdGNHeGxJR1J6YVNC
          allYUmhJR1ZlWTNKNWNIUmxaQ0IzYVhSb0lFRkZVekV5T0NCclpYa2dabkp2Y1NC
          eVpXCI6ICAgICAgIE5wY0dsbGJuUnpMbVZlWTNKNWNIUmxaRjlyWlhrIiwKICAgICAg
          InRhZyI6ICJjMkZ0Y0d4bElHRjFkR2hsYm5ScFkyRjBhVzllSUhSaFp3IogogICAg
          fQogIH0KfQ",
        "protected": "eyJhbGciOiJSUzI1NiJ9",
        "signature": "c2FtcGx1IHNPZ25hdHVyZQ"
      }
    }
  ]
}

```

The TEE returns "GetDeviceStateResponse" back to the OTrP Broker, which returns message back to the TAM.

A.1.2. Sample CreateSD

A.1.2.1. Sample CreateSDRequest

```
{
  "CreateSDTBSRequest": {
    "ver": "1.0",
    "rid": "req-01",
    "tid": "tran-01",
    "tee": "SecuriTEE",
    "nextdsi": "false",
    "dsi": "Iu-c0-fGrpMmzbbttiWI1U8u7wMJE7IK8wkJpsVuf2js",
    "content": {
      "spid": "bank.com",
      "sdname": "sd.bank.com",
      "spcert": "MIIDFjCCAn-
gAwIBAgIJAIk0Tat0tquDMA0GCSqGSIb3DQEBAQUAMGwxCzAJBgNVBAYTAkTAMQ4wD
AYDVQQIDAVTZW91bDESMBAGA1UEBwwJR3Vyby1kb25nMRAwDgYDVQQKDAdTb2xhY21l
hMRAwDgYDVQQLDAdTb2xhY21lMRUwEwYDVQQDDAxTb2xhLWNpYS5jb20wHhcNMTUwN
zAyMDg1MTU3WhcNMjAwNjMwMDg1MTU3WjBzMQswCQYDVQQGEWJLUjEOMAwGA1UECAw
FU2VvdWwxEjAQBgNVBAcMCUd1cm8tZG9uZzEQMA4GA1UECgwHU29sYWNpYTEQMA4GA
1UECwwHU29sYWNpYTEVMBMGGA1UEAwwMU29sYS1jaWEuY29tMIGfMA0GCSqGSIb3DQE
BAQUAAAGNADCBiQKBgQDYWLRff2OFMEciwSYsyhaLY4kslaWcXA0hCWJRaFzt5mU-
lpSJ4jeu92inBbsXcI8PfRbaItsgW1TD1Wg4gQH4MX_YtaBoOepE--
3JoZZyPyCWS3AaLYWrDmqFXdbza01i8GxB7zz0gWw55bZ9jyzcl5gQzWSqMRpx_dca
d2SP2wIDAQABO4G_MIG8MIGGBgNVHSMEfzB9oXCkbjBzMQswCQYDVQQGEWJLUjEOMAw
GA1UECAwFU2VvdWwxEjAQBgNVBAcMCUd1cm8tZG9uZzEQMA4GA1UECgwHU29sYWNp
YTEQMA4GA1UECwwHU29sYWNpYTEVMBMGGA1UEAwwMU29sYS1jaWEuY29tggkAiTRNq3
S2q4MwCQYDVDR0TBAlwADAQBgNVHQ8BAf8EBAMCBsAwFgYDVDR0LAQH_BAwwCgYIKwYB
BQUHAWMwDQYJKoZIhvcNAQEFBQADgYEAfMhRwEQ-
LDa907P1N0mcLORpo6fW3QuJfuXbRQRQGoXddXMKazI4VjbGaXhey7Bzvk6TZYDa-
GRiZby1J47UPaDQR3UiDzVvXwCOU6S5yUhNJsW_BeMViYj4lssX28iPpNwLUCVm1QV
THILI6afLCRWXXclcl1L5KGY290OwIdQ",
      "tamid": "TAM_x.acme.com",
      "did": "zAHkb0-SQh9U_OT8mR5dB-tygcqpUJ9_x07pIiw8WoM"
    }
  }
}
```

Below is a sample message after the content is encrypted and encoded

```
{
  "CreateSDRequest": {
    "payload": "
eyJDCmVhdGVTRFRFCU1JlcXVlc3QiOmsidmVyIjojMS4wIiwicmlkIjoicmVxLTAxIiwidG
lkIjojdHJhbi0wMSIsInRlZSI6IlNlY3VyYVFRFRSIsIm5leHRkc2kiOiJmYWxzZSIsImRz
aWhhc2giOiIyMmVmOWNkM2U3YzZhZTkzMjZjZGI2ZWQ4OTYyMzU1M2NiYmJjMGMyNDRLYz
gyYmNjMjQyNjliMTViOWZkYTNiIiwuY29udGVudCI6eyJwcm90ZWNOZWQiOiJlLUtBbkdW
dVktS0FuVHJpZ0p4Qk1USTRRMEpETFVoVE1qVTI0b0NkZlEiLCJyZWNPcGllbnRzIjpbey
JoZWfkZXIiOmsiYXNjIjojU1NBMV81In0sImVuY3J5cHRlZF9rZXkiOiJTuZuE2NTl4Q2FJ
cldUeUlsVTZPLUVsZzU4UUhvT1pCekxVRGptVG9vanBaWE54TVpBakRMcWtaSTdEUzhOVG
FIWHcxczFvZjgydVhsM0d6NlVWMkRoZDZJR2l6Y2VEDGtXclRwZDg4QVYwaWpEYTNXa3lk

```

dEpSVmlPOGdkSlEtV29NSUVJRUXzVGthblZCb25wQkF4ZHE0ckVMbl9TZl1liaFg4Zm9ub2
gxUVUifv0sIm12IjoiQXhZOERDdERhR2xzYkdsamIzUm9aUSIsImNpcGhlcnRleHQiOiI1
bmVWZXdm55UXprR3hZeWw5QlFrZTJVNjVaOHp4NDdlb3NzM3FETy0xY2FfNEpFY3NLcj
ZhNjF5QzBUb0doYnJOQWJXbVRSemMwsXB5bTF0ZjdGemp4UlhbBaTZBYnVSM2gzSUPRS1Bj
UUVvRUlkZ2tWX0NaZTM2eTBkVDBpRFBMc1g0QzFkb0dmMEDvaWViRC1yVUg1VUteY3BsTW
91tjZvUnFyd0dnNUhxLTJXM3B4MULzY0h4SktRZm11dkYxMTJ4ajBmZFNZX0N2WFE1NTJr
TVRDUW1ZbzRPaGF2R0ZvaG9TZVNaGZSVG1LYWp3OThkTzdhdREdrUEpRU1BtYVHVHw1lEMW
JXd01nMXFRV3RPd19EZ1IyZDNzTzVUN0pQMDJDUfprVXBiQ3dZYVcybW9HN1c2Z1c2U3V5
Q2lpd2pQWmZSqmIzSktTVTFTd1kxYXZvdW02OWctaDB6by12TGZvbHRrWfV2LVdPTXZTY0
JzR25NRzZYZnMzbXlTWnJlWTNRR09wVVRzdjFCQ0JqSTJpdjkw2U2aXFCcVpxQVBxbzdi
ajYwVlJGQzZPT1NLZEXGQTIyU3pqRHo1dmtntXNEaHkwSzlDeVhYN1Z6MkNLTXJvQjNiUE
xZFZF9abtZuVWlkTFN5cVJ5cXJxTmVnN1lmQng3aV93X0dzRW9rX1VYZXd6RGtneHp6RjZj
XzZ6S0s3UFktVnVmYUo0Z2dHZmlpOHEwMm9RZ1VEZTB2Vm1FWDC0c2VQXZRxakVpZVVOYm
xBZE9sS2dBWlFGdEs4dy1xVUMzSzVGTjRoUG9yDeC2b3lPVUpOQTVFZV2Qy1jR2tMcTNQ
UG1GRmQyaUtOTELCTEJzVWl6c1h3RERvZVA5SmktWgt5ZEQtREN1SHdpCno0OEENNwVLSj
Q5WVdqRUtFQko2T01NNUNmZH4cDNmVG1uUTdfTXcwZ3FZVDRiOUJJSnBfwjA3TTctNUPe
emg0czhyU3dsQzFXU3V2RmhRWlJCcXJtX2RaUlRiB0VaZldXc1VCSWVNWWdxNG1zb0JqTj
NXSzhnRWYwZGI5a3Z6UG9LYmpJRy10UUE2R2l1X3pHaFVfLXFBV1lLemVKMDZ6djRIWlBO
dHktQXRyTGF0WGhtUTdOQlVrX0hvbjdOUWxhU1g1ZHVNVmN4bGs1ZHVrWfZNMDgxa09wYV
kzbdliQVffYvHtM0FNAffTTTVsT3dnTDZJazFPYVpaTGfMLUE3eJlITnLESmFEWTVhakZK
TWFdV1lFOG94Y1NoQUktNXA2MmNuT0xzV0dNWNKt1BGVTZpcWlMR19oc3JfN1NKMURhbd
VtQ0YycnBJLUItMlhuckxZR01ZS0NEZ2V2dGFnbilDVUV6RURwR3ozQ2VLcWdQU0Vqd3BK
NOM3NXduYt1CSmtTUKpOdNla3hoWE1rcnNEazRHVVpMSDdQYzFYZHdRTXhxdWpZnmXJSV
EycjM1NWETVkotWHdPcFpfY3RPdW96LTA4WHdYQ3RkTEliSFFVTG40RjLMRTtanU0dUxS
bjNSc043WWZ1S3dCVmVEZDJ6R3NBY0s5SV1Da3hOaDk3dDluYW1iMDZqSXVOWXF5QkhWRU
9nTkhicilrMDY1bW9OVk5lVVUyMm5OdVNKS0ZxVnIxT0dKNGVfNXkzYkNwTmxTeEFPV1Bn
RnJzU0Flc2JJOWw4eVJtVTAwenJYdGc4OWt5SjlCcXN2eXA1RE8wX2FtS1JyMXB1MVJVWF
lFZzB2ampKS1FSdDVZbXRUNFJzaWpzdGRDWDg3UUxJaUdSY0hDdlJzUzZSdJESmNYR1ht
UGQyc0ZmNUZyNnJnMkFzX3BmUHN3cnF1WlAxbVFLc3RPMFVktXpqMTlyb2N1NHVxVXlHUD
lWWU54cHVnWVdNSjRYb1dRelJtWGNTUEJ4VETnenFPS2s3UnRzWWVMNX14LVM4NjV0cHVz
dTA0bXpzYUJRZ21od1ZFVXBRdWNrcG1YWkNLNH1JUXktaHNFQUlJSmVxdFB3dVAySXF0X2
I5dlk0bzExeXdzeXhZdmp2RnKN0VVZU1MaGE2R2dSanBSbnU5RWIzRn1JZ0U5M0VVNEEW
T0lUMWlOSGNRYWc0eWtOc3dPdkxQbjZIZ21zQ05ESlgwekc2RlFDMTZRdjBSQ25SVTdfV2
VvblhSTUZWUzZRZ1JiSk45R1NMckN5bklJSWxUCDBxNHBaS05zM0tqQ2tMUzJrb3Bhd2Y0
WF9BU1lmTko3a0s5eW5BR0dCcKtnUWJNRWVxUEFmMDBKM1YtVXpuU1JMzmQ4SGs3Y2JEdk
5RQlhHQW9BR0ViaGrwVUC0RXFwMlVyQko3dEtyUUVSR1h4RTVsOFNHY2czQ1RmN2Zoazdx
VEFBVjVsWEFnoUtoUDF1clZRZk1fUlBleHfNTG9WQVVKV2syQkF6WF9uSEhkVVhaSVBIOG
hLeDctdEFRV0dTWUd0R2FmanZJZzI2c082TzloQWZVd3BpSV90MzF6SkZORDU0OTZURHBz
QmNnd2dMLU1UcVhCRUJ2NEhvQld5SG1DVjVFMUwiLCJ0YWciOiJkbXlEeWZJVlNjU1Ren
ExOEgybFRiEEMxbl9HZEtrdnZNMDJUChdsYzQwIn19fQ",
"protected": "e-KAnGFsZ-KAnTrigJxSUzI1NuKAnX0", //RSAwithSHA256
"header": {
 "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d",
 "signer": "
 MIIC3zCCAkigAwIBAgIJAJf2ffkE1BYOMA0GCSqGSIB3DQEBBQUAMF0xZAJBgNVBA
 YTA1VTMRMwEQYDVQIDApDYWxpZm9ybmlhMRMwEQYDVQIDApDYWxpZm9ybmlhMSEw
 HwYDVQQKBHJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQwHhcNMTUwNzAyMDkwMTE4Wh
 cNMjAwNjMwMDkwMTE4WjBaMQswCQYDVQQGEwJVUzETMBEGA1UECAwKQ2FsaWZvcml5p

```

    YTETMBEGA1UEBwwKQ2FsaWZvcn5pYTEhMB8GA1UECgwYSW50ZXJuZXQgV2lkZ2l0cy
    BQdHkgTHRkMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC8ZtxM1bYickpgSVG-
    meHInI3f_chlMBdL8l7daOEztSs_a6GLqmvSu-
    AoDpTsfEd4EazdMBp5fmgLRGdCYMcI6bgpO94h5CCnlj8xFKPq7qGixdwGUA6b_ZI3
    c4cZ8eu73VMNrrn_z3WTZlExlpT9XVj-
    ivhfJ4a6T20EtMM5qwIDAQABo4GsMIGpMHQGA1UdIwRtMGuhXqRcMFoxCzAJBgNVBA
    YTA1VTMRMwEQYDVQQIDApDYWxpZm9ybmlhMRMwEQYDVQQHDApDYWxpZm9ybmlhMSEw
    HwYDVQQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGSCCQCX9nxZBNQWDjAJBgNVHR
    MEAjAAMA4GA1UdDwEB_wQEAwIGDAWBgNVHSUBAf8EDDAKBggrBgEFBQcDAzANBgkq
    hkiG9w0BAQUFAAOBgQAGkz9QpoxghZUWT4ivem4cIckfxzTBBiPHCjrrjB2X8Ktn8G
    SZlMdyIZV8fwdEmD90IvtMHgtzK-
    9wo6Aibj_rVIpXGb7trP82uzc2X8VwYnQbuqQyzofQvcwZHLyplvi95pZ5fVrJvnYA
    UBFyfrdT5GjqLlnqH3a_Y3QPscuCjg"
  },
  "signature": "nuQUsCTEBLeaRzuwd7q1iPIYEJ2eJfur05sT5Y-
  N03zFRcv1jvrqMHTx_pw0Y9YWjpmoWfpfelhwGEko9SgeeBnznmkZbp7kjs6MmX4CKz
  9OApe3-VI7yL9Yp0WNdRh3425eYfuapCy3lcXFln5JBAUnU_OzUg3RWxcU_yGnFsw"
}
}

```

A.1.2.2. Sample CreateSDResponse

```

{
  "CreateSDTBSResponse": {
    "ver": "1.0",
    "status": "pass",
    "rid": "req-01",
    "tid": "tran-01",
    "content": {
      "did": "zAHkb0-SQh9U_OT8mR5dB-tygcqpUJ9_x07pIiw8WoM",
      "sdname": "sd.bank.com",
      "teespaik": "AQABjY9KiwH3hkMmSAAN6CLXot525U85WNlWKAQz5TOdfe_CM8h-
      X6_EHX1gOXoyRXaBiKmqWb0YZLCABTwlytdXy2kWa525imRho8Vqn6HDGsJDZPDru9
      GnZR8pZX5ge_dWXB_uljMvDttc5iAWEJ8ZgcpLGtBTGLZnQoQbjtn1lIE",
    }
  }
}

```

Below is the response message after the content is encrypted and encoded.

```

{
  "CreateSDResponse": {
    "payload": "
    eyJDcmVhdGVTRFRCU1Jlc3BvbmlIjp7InZlciI6IjEuMCIsInN0YXR1cyI6InBhc3Mi
    LCJyaWQiOiJyZXEtMDEiLCJ0aWQiOiJ0cmFuLTAxIiwIY29udGVudCI6eyJwcm90ZWNO
    ZWQiOiJlLUtBbkdwVktS0FuVHJpZ0p4Qk1USTRMEpETFVoVE1qVTI0b0NkZ1EiLCJy
    ZWNpcGllbnRzIjpbeyJoZWZkZXIiOiJ0c3R5bWV81In0sImVuY3J5cHRlZF9r

```



```

ZXkiOiJOX0I4R3pldUlfN2hwd0wwTFpHSTkxVWVBbmXJRkJfcndmZU1yZERrWnFGak1s
VVhjd1I0XzhhOGhyeFI4SXR3aEtFZnVfRWVLRDBQb0dqQ2pCSHcxG1ULUN6eWhsbW5v
Slk3LXl1WnZzRkRpc2VNTkd0eGE0OGZJYUs2VWx5NUZMYXBCZVc5T1I5bmktOU9GQV9j
aFVuWW13b2Q4ZTJFa0Vpd0JEZ1EzMk0ifV0sIm12IjoiQXhZOERDdERhR2xzYkdsamIz
Um9aUSIsImNpcGhlcnRleHQiOiJsalh6Wk5JTmR1WjFaMXJHVElktjBiVUp1RDRVV2xT
QVptLWd6YnJINFVDYy1jMEFQenMtMwDWSFk4NTRUR3VMYkdyRmVHcDFqM2Fsb1lacWZp
ZnE4aEt3Ty16RF1BN2tmVFhBZHp6czM4em9xeG4zbHoyM2w1RU1GUWhrOHBRWTRYTHRw
M3ZBQWlnYn1rQ1Q3VS1CWDdWcjbacVNhYWZTQVZ4OFBLQ1RIU3hHN3hHVko0NkxxRzJS
RE54WXQ4RC1SQ3lZUi1zRTM0MUFKZldeC2FLaGRRbzJXc jNVN1hTOWFqaXJtWjdqTlJ4
cVRodHJBRWl1Y1ctOEJMdVFHWEZ1YUHLMTZrenJKUGl4d0VXbzJ4cmw4cmkwc3ZRCmpl
Z2M3MET2Z0IONUVaNHZiNXR0YlUya25hN185QU1Wcm4wLUJaQ1Bnb280MWlFblhuNVJn
TXY2c2V2Y1JPQ2xHMnpWSjFoRkVLYjk2akeiLCJ0YWciOiIzOTZISTk4Uk1NQnR0eDlo
ZUtsODROaVZld0lJSzI0UET2ZlRGYzFrbeJzIn19fQ",
"protected": "e-KAnGFsZ-KAnTrigJxSUzI1NuKAnX0",
"header": {
  "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d",
  "signer": "
MIIC3zCCAkigAwIBAgIJAJf2ffkE1BYOMA0GCSqGSIb3DQEBBQUAMF0xCzAJ
BgNVBAYTA1VTMRMwEQYDVQQIDApDYWxpZm9ybmlhMRMwEQYDVQQHDApDYWxp
Zm9ybmlhMSEwHwYDVQQKDBhJbnRlcml5dCBXaWRnaXRzIFB0eSBMdGQwHhcN
MTUwNzAyMDkwMTE4WWhcNMjAwNjMwMDkwMTE4WjBaMQswCQYDVQQGEWJVUzET
MBEGA1UECAwKQ2FsaWZvcml5pYtETMBEGA1UEBwwKQ2FsaWZvcml5pYtEhMB8G
A1UECgwYSW50ZXJuZXQgV2lkZ2l0cyBQdHkgTHRkMIGfMA0GCSqGSIb3DQEB
AQUAA4GNADCBiQKBgQC8ZtxM1bYickpgSVG-
meHInI3f_ch1MBdL8l7daOEztSs_a6GLqmvSu-
AoDpTsfEd4EazdMBp5fmgLRGdCYMcI6bgp094h5CCnlj8xFKPq7qGixdwGUA
6b_ZI3c4cz8eu73VMNrrn_z3WTZlExlpT9XVj-
ivhfJ4a6T20EtMM5qwIDAQAB04GsMIGpMHQGA1UdIwRtMGUhxQRCMF0xCzAJ
BgNVBAYTA1VTMRMwEQYDVQQIDApDYWxpZm9ybmlhMRMwEQYDVQQHDApDYWxp
Zm9ybmlhMSEwHwYDVQQKDBhJbnRlcml5dCBXaWRnaXRzIFB0eSBMdGSCCQCX
9nxZBNQWDjAJBgNVHRMEA jAAMA4GA1UdDwEB_wQEAWIGDAWBgNVHSUBAf8E
DDAKBggrBgEFBQcDazANBgkqhkiG9w0BAQUFAAOBgQAGkz9QpoxghZUWT4iv
em4cIckfxzTBBiPHCjrrjB2X8Ktn8GSZ1MdyIZV8fwdEmD90IvtMHgtzK-
9wo6Aibj_rVipxGb7trP82uzc2X8VwYnQbuqQyzofQvcwZHLyplvi95pZ5fv
rJvnYAUBFyfrdT5GjqLlnqH3a_Y3QPscuCjg"
},
"signature": "jnJtaB0vFFwrE-qKOR3Pu9pf2gNoI1s67GgPCTq0U-
qrz97svKpuh32WgCP2MwCoQPEswsEX-nxhIx_siTe4zIPO1nBYn-
R7b25rQaF8708uAOOnBN5Y12Jk3laIbs-
hGE32aRZDhrVoyEdSvIFrT6AQqD20bIAZGqTR-za-900"
}
}

```

A.1.3. Sample UpdateSD

A.1.3.1. Sample UpdateSDRequest

```

{
  "UpdateSDTBSRequest": {
    "ver": "1.0",
    "rid": "1222DA7D-8993-41A4-AC02-8A2807B31A3A",
    "tid": "4F454A7F-002D-4157-884E-B0DD1A06A8AE",
    "tee": "Primary TEE ABC",
    "nextdsi": "false",
    "dsihash":
    "
    IsOvwpzDk8Onw4bCrsKTJsONwrbDrcKJYjVTw4vCu8OAw4JEw6zCgsK8w4JCacKxW8Kf
    w5o7",
    "content": { // NEEDS to BE ENCRYPTED
      "tamid": "id1.TAMxyz.com",
      "spid": "com.acmebank.spid1",
      "sdname": "com.acmebank.sdname1",
      "changes": {
        "newsdname": "com.acmebank.sdname2",
        "newspid": "com.acquirer.spid1",
        "spcert":
        "MIIDFjCCAn-
        gAwIBAgIJAik0Tat0tquDMA0GCSqGSIB3DQEBBQUAMGwxCzAJBgNVBAYTAkTAMQ4
        wDAYDVQQIDAVTZW91bDESMBAGAlUEBwwJR3Vyby1kb25nMRAwDgYDVQQKDAkTb2x
        hY2lhMRAwDgYDVQQLDAdTb2xhY2lhMRUwEwYDVQQDDAxTb2xhLWNpYS5jb20wHhc
        NMTUwNzAyMDg1MTU3WhcNMjAwNjMwMDg1MTU3WjBsMQswCQYDVQQGEWJLUjEOMAw
        GA1UECAwFU2VvdWwxEjAQBgNVBAcMCUdlcm8tZG9uZzEQMA4GA1UECgwHU29sYWN
        pYTEQMA4GA1UECwwHU29sYWNpYTEVMBMGAlUEAwWUMU29sYS1jaWEuY29tMIGfMA0
        GCSqGSIB3DQEBQUAA4GNADCBiQKBgQDYWLrFf2OFMEciwSYsyhaLY4kslaWcXA0
        hCWJRaFzt5mU-
        lpSJ4jeu92inBbsXcI8PfRbaItsgW1TD1Wg4gQH4MX_YtaBoOepE--
        3JoZZyPyCWS3AaLYWrDmqFXdbzaO1i8GxB7zz0gWw55bZ9jyzcl5gQzWSqMRpx_d
        cad2SP2wIDAQABo4G_MIG8MIGGBgNVHSMefzB9oXCkbjBsMQswCQYDVQQGEWJLUj
        EOMAwGA1UECAwFU2VvdWwxEjAQBgNVBAcMCUdlcm8tZG9uZzEQMA4GA1UECgwHU2
        9sYWNpYTEQMA4GA1UECwwHU29sYWNpYTEVMBMGAlUEAwWUMU29sYS1jaWEuY29tgg
        kAiTRNq3S2q4MwCQYDVROTBAlwADA0BgNVHQ8BAf8EBAMCBsAwFgYDVRO1AQH_BA
        wwCgYIKwYBBQUHAWMwDQYJKoZIhvcNAQEFBQADgYEAfMhRwEQ-
        LDa9O7P1N0mcLORpo6fW3QuJfuXbRQRQGoXddXMKazI4VjbGaXhey7Bzvk6TZYDa
        -
        GRiZby1J47UPaDQR3UiDzVvXwCOU6S5yUhNJsW_BeMViYj4lssX28iPpNwLUCVm1
        QVTHILI6afLCRWXXclclL5KGY2900wIdQ",
        "renewteespaik": "0"
      }
    }
  }
}

```

A.1.3.2. Sample UpdateSDResponse

```
{
  "UpdateSDTBSResponse": {
    "ver": "1.0",
    "status": "pass",
    "rid": "1222DA7D-8993-41A4-AC02-8A2807B31A3A",
    "tid": "4F454A7F-002D-4157-884E-B0DD1A06A8AE",
    "content": {
      "did": "MTZENTE5Qzc0Qzk0NkUxMzYxNzk0NjY4NTc3OTY4NTI=",
      "teespaik":
        "AQABjY9Kiwh3hkMmSAAN6CLXot525U85WNlWKAQz5TOdfe_CM8h-
        X6_EHX1gOXoyRXaBiKMqWb0YZLCABTwlytdXy2kWa525imRho8Vqn6HDGsJDZPDru9
        GnZR8pZX5ge_dWXB_uljMvDttc5iAWEJ8ZgcpLGtBTGLZnQoQbjtn1lIE",
      "teespaiktype": "RSA"
    }
  }
}
```

A.1.4. Sample DeleteSD

A.1.4.1. Sample DeleteSDRequest

The TAM builds message - including data to be encrypted.

```
{
  "DeleteSDTBSRequest": {
    "ver": "1.0",
    "rid": "{712551F5-DFB3-43f0-9A63-663440B91D49}",
    "tid": "{4F454A7F-002D-4157-884E-B0DD1A06A8AE}",
    "tee": "Primary TEE",
    "nextdsi": "false",
    "dsihash": "AAECAwQFBgcICQoLDA0ODwABAgMEBQYHCAkKCwwNDg8=",
    "content": ENCRYPTED {
      "tamid": "TAM1.com",
      "sdname": "default.acmebank.com",
      "deleteta": "1"
    }
  }
}
```

The TAM encrypts the "content".

```

{
  "DeleteSDTBSRequest": {
    "ver": "1.0",
    "rid": "{712551F5-DFB3-43f0-9A63-663440B91D49}",
    "tid": "{4F454A7F-002D-4157-884E-B0DD1A06A8AE}",
    "tee": "Primary TEE",
    "nextdsi": "false",
    "dsihash": "AAECAwQFBgcICQoLDA0ODwABAgMEBQYHCAkKCwwNDg8=",
    "content": {
      "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
      "recipients": [
        {
          "header": {
            "alg": "RSA1_5"
          },
          "encrypted_key":
            "QUVTMTI4IChDRUspIGtleSwgZW5jcnlwdGVkIHdpdGggVFNNIFJTQSBwdWJsaWMga2V5LCB1c2luZyBSU0ExXzUgcGFkZGluZW"
        }
      ],
      "iv": "rW05DVmQX9ogelMLBIogIA",
      "ciphertext":
        "c2FtcGx1IGRzaSBkYXRhIGVuY3J5cHRlZCB3aXRoIEFFUzEyOCBrZXkgZnJvbSB5ZWNPcGllbnRzLmVuY3J5cHRlZF9rZXk",
      "tag": "c2FtcGx1IGF1dGhlbnRpY2F0aW9uIHRhZW"
    }
  }
}

```

The TAM signs the "DeleteSDTBSRequest" to form a "DeleteSDRequest"

```

{
  "DeleteSDRequest": {
    "payload": "
ewoJIkRlbGV0ZVNEVEJlTUUmVxdWVzdCI6IHsKCQkidmVyIjogIjEuMCIsCgkJInJp
ZCI6ICJ7NzEyNTUxRjUtREZCMY00M2YwLTlBNjMtNjYzNDQwQjkxRDQ5fSIsCgkJ
InRpZCI6ICJ7NEY0NTRBN0YtMDAyRC00MTU3LTg4NEUtQjBERDFBMDZBOEFFfSIs
CgkJInRlZSI6ICJQcm1tYXJ5IFRFRSIsCgkJIm5leHRkc2kiOiAiZmFsc2UiLAoJ
CSJkc2loYXNoIjogIkFBRUNBdlFGQmdjSUNRb0xEQTBPRHdBQkFnTUUVUUVlIQ0Fr
S0N3d05EZzg9IiwKCQkiY29udGVudCI6IHsKCQkKJInByb3RlY3RlZCI6ICJleUps
YmlNaU9pSkJNVEk0UTBKRExVaFRNaUySW4wIiwKCQkKJInJlY2lwaWVudHMiOiBb
ewoJCQkKJImh1YWRlciI6IHsKCQkKJCQkiYWxnIjogIlJTQTFfNSIKCQkKJCX0sCgkJ
CQkiZW5jcnldGVkX2tleSI6ICJRVVZUTVRJNElDaERSVXNwSUd0bGVtd2daVzVq
Y25sd2RHVmtJSGRwZEbnZlZGt5JRkpUUVNCd2RXSnNhV0lnYTJWNUxkQjFjMmx1
WnlCU1UwRXhYelVnY0dGalpHbHVadyIKCQkKJfV0sCgkJCSJpdiiI6ICJyV081RFZt
UVg5b2dlbE1MQklvZ0lBIiwKCQkKJImNpcGhlcnRleHQiOiAiYzJGdGNHeGxJR1J6
YVNCa1lYUmhJR1ZlWTNKNWNIUmxaQ0IzYVhSb0lFRkZVekV5T0NCclpYa2dabkp2
YlNCeVpXTnBjR2xsYm5SekxtVnVZM0o1Y0hSbFpGOXJaWGsilaAoJCQkidGFniJog
ImMyRnRjR3hsSUdGMWRHaGxiblJwWTJGMGFkXOXVJSFJoWnciCgkJfQoJfQp9",
    "protected": "eyJhbGciOiJSUzI1NiJ9",
    "header": {
      "x5c": ["ZXhhbXBsZSBBU04xIHNPZ251ciBjZXJ0aWZpY2F0ZQ==",
        "ZXhhbXBsZSBBU04xIENBIGNlcnRpZmljYXRl"]
    },
    "signature": "c2FtcGx1IHNPZ25hdHVyZQ"
  }
}

```

A.1.4.2. Sample DeleteSDResponse

The TEE creates a "DeleteSDTBSResponse" to respond to the "DeleteSDRequest" message from the TAM, including data to be encrypted.

```

{
  "DeleteSDTBSResponse": {
    "ver": "1.0",
    "status": "pass",
    "rid": "{712551F5-DFB3-43f0-9A63-663440B91D49}",
    "tid": "{4F454A7F-002D-4157-884E-B0DD1A06A8AE}",
    "content": ENCRYPTED {
      "did": "MTZENTE5Qzc0Qzk0NkUxMzYxNzk0NjY4NTc3OTY4NTI=",
    }
  }
}

```

The TEE encrypts the "content" for the TAM.

```

{
  "DeleteSDTBSResponse": {
    "ver": "1.0",
    "status": "pass",
    "rid": "{712551F5-DFB3-43f0-9A63-663440B91D49}",
    "tid": "{4F454A7F-002D-4157-884E-B0DD1A06A8AE}",
    "content": {
      "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0K",
      "recipients": [
        {
          "header": {
            "alg": "RSA1_5"
          },
          "encrypted_key":
            "QUVTMTI4IChDRUspIGtleSwgZW5jcnlwdGVkIHdpdGggVFNNIFJTQSBwdWJsaWMg
            a2V5LCB1c2luZyBSU0ExXzUgcGFkZGluZW"
        }
      ],
      "iv": "ySGmfZ69YlcEilNr5_SGbA",
      "ciphertext":
        "c2FtcGx1IGRzaSBkYXRhIGVuY3J5cHRlZCB3aXRoIEFFUzEyOCBrZXkgZnJvbSB5ZW
        NpcGllbnRzLmVuY3J5cHRlZF9rZXk",
      "tag": "c2FtcGx1IGFldGhlbnRpY2F0aW9uIHRhZW"
    }
  }
}

```

The TEE signs "DeleteSDTBSResponse" to form a "DeleteSDResponse"

```

{
  "DeleteSDResponse": {
    "payload": "
ewoJIkRlbGV0ZVNEVEJtUmVzcG9uc2UiOiB7CgkJInZlciI6ICIxLjAiLAoJCSJz
dGF0dXMiOiAicGFzcyIsCgkJInJpZCI6ICJ7NzEyNTUxRjUtREZCMY00M2YwLTlB
NjMtNjYzNDQwQjkxRDQ5fSIsCgkJInRpZCI6ICJ7NEY0NTRBN0YtMDAyRC00MTU3
LTg4NEUtQjBERDFBMDZBOEFFfSIsCgkJImNvbnRlbnQiOiB7CgkJCSJwcm90ZWNO
ZWQiOiAiZXlKbGJtTWlPaUpCTVRJNFewSkRMVWhUTWpVMkluMEsiLAoJCQkicmVj
aXBpZW50cyI6IFt7CgkJCQkiaGVhZGVyIjogewoJCQkJCSJhbGciOiAiUlNBMV81
IgoJCQkJfSwKCQkJCSJlbnNyeXB0ZWRfa2V5IjogIlFVVlRNVEk0SUNoRFJvc3BJ
R3RsZVN3ZlpxNWpjbmx3ZEdWa0lIZHBkR2dnVkJOTklGS1RRU0J3ZFdKc2FXTWdh
MlY1TENCMWMybHVaeUJTVTBFeFh6VWdjR0ZrWkdsdVp3IgoJCQl9XSwwKCQkJIml2
IjogInlTR2lmWjY5WWxjRWlsTnI1X1NHYkEiLAoJCQkiY2lwaGVydGV4dCI6ICJj
MkZ0Y0d4bElHUUnphU0JrWVhSaElHVnVZM0o1Y0hSbFpDQjNhWFJvSUVGRlV6RXlP
Q0JyWlhrZlpuSnZiU0J5WldOcGNHbGxib1J6TG1WdVksSjVjSFJsWkY5clpYayIs
CgkJCSJ0YWciOiAiYzJGdGNHeGxJR0YxZEdobGJuUnBZMkYwYVc5dUllUmhadyIK
CQl9Cgl9Cn0",
    "protected": "eyJhbGciOiJSUzI1NiJ9",
    "signature": "c2FtcGxlIHNPZ25hdHVyZQ"
  }
}

```

The TEE returns "DeleteSDResponse" back to the OTrP Broker, which returns the message back to the TAM.

A.2. Sample TA Management Messages

A.2.1. Sample InstallTA

A.2.1.1. Sample InstallTAResponse

```

{
  "InstallTATBSRequest": {
    "ver": "1.0",
    "rid": "24BEB059-0AED-42A6-A381-817DFB7A1207",
    "tid": "4F454A7F-002D-4157-884E-B0DD1A06A8AE",
    "tee": "Primary TEE ABC",
    "nextdsi": "true",
    "dsihash":
    "
    IsOvwppzDk8Onw4bCrsKTJsONwrbDrcKJYjVTw4vCu8OAw4JEw6zCgsK8w4JCacKxW8Kf
    w5o7",
    "content": {
      "tamid": "idl.TAMxyz.com",
      "spid": "com.acmebank.spid1",
      "sdname": "com.acmebank.sdname1",
      "taid": "com.acmebank.taid.banking"
    },
    "encrypted_ta": {
      "key":
      "mLBjodcE4j36y64nC/nEs694P3XrLAOokjisXIGfs0H7lOEmT5FtaNDYEMcg9RnE
      ftlJGHO7N0lgcNcjoXBmeuY9VI8xzrsZM9gzH6VBKtVONSx0aw5IAFkNcyPZwDdZ
      MLwhvrzPJ9Fg+bZtrCoJz18PUz+5aNl/dj8+NM85LCXXcBlZF74btJer1Mw6ffzT
      /grPiEQTeJ1nEm9F3tyRsvctInsnPJ3dEXv7sJXMrhRKAeZsqKzGX4eiZ3rEY+FQ
      6nXULC8cAj5XTKpQ/EkZ/iGgS0zcXR7KUJv3wFEmtBtPD/+ze08NILLmxM8olQFj
      //Lq0gGtq8vPC8r0oOfmbQ==",
      "iv": "4F5472504973426F726E496E32303135",
      "alg": "AESCBC",
      "ciphertadata":
      ".....0x/5KGCXWfg1Vrjm7zPVZqtYZ2EovBow+7EmfOJ1tbk.....=",
      "cipherpdata": "0x/5KGCXWfg1Vrjm7zPVZqtYZ2EovBow+7EmfOJ1tbk="
    }
  }
}

```

A.2.1.2. Sample InstallTAResponse

A sample to-be-signed response of InstallTA looks as follows.

```

{
  "InstallTATBSResponse": {
    "ver": "1.0",
    "status": "pass",
    "rid": "24BEB059-0AED-42A6-A381-817DFB7A1207",
    "tid": "4F454A7F-002D-4157-884E-B0DD1A06A8AE",
    "content": {
      "did": "MTZENTE5Qzc0Qzk0NkUxMzYxNzk0NjY4NTc3OTY4NTI=",
      "dsi": {
        "tfwdata": {

```



```

    "tbs": "ezRGNDU0QTdGLTAwMkQtNDE1Ny04ODRFLUIwREQxQTA2QThBRX0="
    "cert": "ZXhhbXBsZSBGVyBjZXJ0aWZpY2F0ZQ==",
    "sigalg": "U1MyNTY=",
    "sig": "c2FtcGx1IEZXIHNPZ25hdHVyZQ=="
  },
  "tee": {
    "name": "Primary TEE",
    "ver": "1.0",
    "cert": "c2FtcGx1IFRFRSBjZXJ0aWZpY2F0ZQ==",
    "cacert": [
      "c2FtcGx1IENBIGNlcnRpZmljYXRlIDE=",
      "c2FtcGx1IENBIGNlcnRpZmljYXRlIDI="
    ],
    "sdlist": {
      "cnt": "1",
      "sd": [
        {
          "name": "com.acmebank.sdname1",
          "spid": "com.acmebank.spid1",
          "talist": [
            {
              "taid": "com.acmebank.taid.banking",
              "taname": "Acme secure banking app"
            },
            {
              "taid": "acom.acmebank.taid.loyalty.rewards",
              "taname": "Acme loyalty rewards app"
            }
          ]
        }
      ]
    }
  },
  "teeaiklist": [
    {
      "spaik":
        "c2FtcGx1IEFTTjEgZW5jb2RlZCBQS0NTMSBwdWJsaWNrZXk=",
      "spaiktype": "RSA",
      "spid": "acmebank.com"
    }
  ]
}

```

A.2.2. Sample UpdateTA

A.2.2.1. Sample UpdateTAResponse

```

{
  "UpdateTATBSRequest": {
    "ver": "1.0",
    "rid": "req-2",
    "tid": "tran-01",
    "tee": "SecuriTEE",
    "nextdsi": " false",
    "dsihash": "gwjul_9MZks3pqUSN1-eLlaViwGXNAxk0AIKW79dn4U",
    "content": {
      "tamid": "TAM1.acme.com",
      "spid": "bank.com",
      "sdname": "sd.bank.com",
      "taid": "sd.bank.com.ta"
    },
    "encrypted_ta": {
      "key":
      "
      XzmAn_RDV3IozMwNWhiB6fmZlIs1YUvMKlQAv_UDoZ1fvGGsRGo9bT0A440aYMgLt
      GilKypoJjCgiJdaHgamaJgRSc4Je2otpnEEagsahvDNoarMCC5nGQdkRxW7Vo2NKG
      L892HGeHkJVshYmlcU1FQ-BhiJ4NAykFwlqC_oc",
      "iv": "AxY8DCtDaGlsbGljb3RoZQ",
      "alg": "AESCBC",
      "ciphernewtadata":
      "KHqOxGn7ib1F_14PG4_UX9DBjOcWkiAZhVE-U-
      67NsKryHGokeWr2spRWfdU2KWaaNncHoYGWetbCH7XyNbOFh28nzwUmstep4nHWbAl
      XZYTnKENCABPpuw_G3I3HADo"
    }
  }
}

{
  "UpdateTAResponse": {
    "payload":
    "
    eyJVCGRhdGVUQVRCU1JlcXVlc3QiOnsidmVyIjoiMS4wIiwicmlkIjoiemVxLTiiLCJ0
    aWQiOiJ0cmFuLTAxIiwidGVlIjoiU2VjdXJpVEVFfiwibmV4dGRzaSI6ImZhbHN1Iiw
    iZHNpaGFzaCI6Imd3anVsXzlnNWmtzM3BxVVNOMS1lTDFhVml3R1hOQXhrMEFJS1c3O
    WRuNFUiLCJjb250ZW50Ijp7InByb3RlY3RlZCI6ImV5SmxibU1pT2lKQk1USTRRMEp
    ETFVoVE1qVTJJbjAiLCJyZWNPcGllbnRzIjpbeyJoZWZkZXIiOnsiYWxnIjoiU1NB
    MV81In0sImVuY3J5cHRlZGF9rZXkiOiJYem1Bbl9SRFZrM0lvek13TldoaUI2Zmlab
    ElzMVlVdk1LbFFBdl9VRG9aMWZ2R0dzUkdvOWJUMEE0NDBhWU1nTHRHaWxLeXBv
    SmpDZ2lqZGFIZ2FtYUpnU1NjNEplMm90cG5FRWFnc2FodkROb2FyTUNDNW5HUWRr
    UnhXN1ZvMk5LZ0xBODkySEdlSGtKVnNoWW0xY1VsRlEtQmhpSjROQXlrRndscUNfb
    2MifV0sIm12IjoiQXhZOERDdERhR2xzYkdsamIzUm9aUSIsImNpcGhlcnRleHQiOi
    JIYTcwVXRZVetWQmtXRFJuMi0w
  "
  }
}

```

```

SF9IdkZtazl5SGtoVV91bk1OLWc1T3BqLWF1NGFUb21xWklMYzVzYTdENnZSjF6eW04
QWlJOEJIVXFqc2l5Z0tOcC1HdURJUjFzRXc0a2NhMVQ5ZENuU0RydHhSUFhESVdrZmt3
azZlRlNQWiIsInRhZyI6Im9UN01UTE41eWtBTfBoTDR0aUh6T1pPTGVFeU9xZ0NWaEM5
MXpkclldMU0UifSwiZW5jcnlwdGVkX3RhIj7ImtleSI6Ilh6bUFuX1JEVmszSW96TXdO
V2hpQjZmbVpsSXMxWV2TUtSUUF2X1VEbl0xZnZHR3NSR285Y1QwQTQ0MGFZTWdMdEdp
bEt5cG9KakNnaWpkYUhnYW1hSmdSU2M0SmUyb3RwbkVfYWdzYW1h2RE5vYXJNQM1bkdR
ZGtSeFc3Vm8yTktnTEE4OTJIR2Vla0pWc2hZbTFjVWxGUS1CaGlKNE5BeWtGd2xxQ19v
YyIsIm12Ijo1QXhZOEERddERhR2xzYkdsamIzUm9aUSIsImFsZyI6IkFFU0NCQyIsImNp
cGhlcm5ld3RhZGF0YSI6IktIcU94R243aW1xR18xNFBHNF9VWD1EQmpPY1draUFaaFZF
LVUtNjdOc0tyeUhb2t1V3Iyc3BSV2ZkVTJLV2FhTm5jSG9ZR3dFdGJDSDDYeU5iT0Zo
MjhuendVbXN0ZXh0bkhXYkFsWFpZVE5rRU5jQUJQcHV3X0czSTNIQURvIn19fQ",
"protected": " eyJhbGciOiJSUzI1NiJ9",
"header": {
  "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d",
  "signer": "
MIIC3zCCAkigAwIBAgIJAJf2fFkE1BYOMA0GCSqGSIb3DQEBAQUAMF0xMzA1BjBBAQ
YTA1VTMRMwEQYDVQIDApDyWxpZm9ybmlhMRMwEQYDVQIDApDyWxpZm9ybmlhMSEw
HwYDVQQKDBhJbnRlcmlldCBXaWRnaXRzIFB0eSBMdGQwHhcNMTUwNzAyMDkwMTE4Wh
cNMjAwNjMwMDkwMTE4WjBaMQswCQYDVQQGEWJVUzETMBEGA1UECAwKQ2FsaWZvcm5p
YTETMBEGA1UEBwwKQ2FsaWZvcm5pYTEhMB8GA1UECgwYSW50ZXJuZXQgV2lkZ2l0cy
BQdHkgTHRkMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC8ZtxM1bYickpgSVG-
meHInI3f_chlMBdL8l7daOEztSs_a6GLqmvSu-
AoDpTsfEd4EazdMBp5fmgLRGdCYMcI6bgpO94h5CCnlj8xFKPq7qGixdwGUA6b_ZI3
c4cZ8eu73VMNrrn_z3WTZlExlpT9XVj-
ivhfJ4a6T20EtMM5qwIDAQABo4GsMIGPMHQA1UdIwRtMGUhxqRcMF0xMzA1BjBBAQ
YTA1VTMRMwEQYDVQIDApDyWxpZm9ybmlhMRMwEQYDVQIDApDyWxpZm9ybmlhMSEw
HwYDVQQKDBhJbnRlcmlldCBXaWRnaXRzIFB0eSBMdGSCCQX9nxZBNQWDjA1BjBBAQ
MEAJAAMA4GA1UdDwEB_wQEAWIGwDAWBgNVHSAf8EDDAKBggrBgEFBQcDAZANBgkq
hkiG9w0BAQUFAAOBgQAGkz9QpoxghZUWT4ivem4cIckfxzTBBiPHCjrrjB2X8Ktn8G
SZlMdyIZV8fwdEmD90IvtMHgtzK-
9wo6Aibj_rVIpxGb7trP82uzc2X8VwYnQbuqQyzofQvcwZHLyplvi95pZ5fVrJvnYA
UBFyfrdT5GjqLlnqH3a_Y3QPscuCjg"
},
"signature": "inB1K6G3EAhF-
FbID83UI25R5Ao8MI4qfrbrmf0UQhjM3O7_g3l6XxN_JkHrGQaZr-
myOkGPVM8BzbUZW5GqxNZwFXwMeaoCjDKc4Apv4WZkd1qKJxkg1k5jaUCfJz1Jmw_XtX
6MHhrLh9ov03S9PtuT1VAQ0FVUB3qFivjSnNU"
}
}

```

A.2.2.2. Sample UpdateTAResponse

```
{
  "UpdateTATBSResponse": {
    "ver": "1.0",
    "status": "pass",
    "rid": "req-2",
    "tid": "tran-01",
    "content": {
      "did": "zAHkb0-SQh9U_OT8mR5dB-tygcqpUJ9_x07pIiw8WoM"
    }
  }
}
```

```

{
  "UpdateTAResponse": {
    "payload": "
eyJVcGRhdGVUQVRCU1Jlc3BvbmlIjP7InZlciI6IjEuMCIsInN0YXR1cyI6InBhc3Mi
LCJyaWQiOiJyZXEtMiIsInRpZCI6InRyYW4tMDEiLCJjb250ZW50IjP7InByb3RlY3Rl
ZCI6ImV5SmxibUlpT2lKQk1USTRRMEpETFVoVE1qVTJjbjAiLCJyZWNPcGllbnRzIjpb
eyJoZWFKZlZlIiOnsiYWxnIjoiUlNBMV81In0sImVuY3J5cHRlZF9rZXkiOiJFaGUxLUJB
UUDJLTNEMFNHdXFGY01MZDJtd0gxQmluRndYQWx1M1FxFVFXZ1RRVm55SUowNFc2MnBK
YWVSREFkeTU0R0FSVjBrVzQ0RGw0MkdUUlhqBE1EZ3BYdXdlWl0clJVV0tNNldCZ2N3
VXVGQTRUR3gwU0I1NTZCd192dnBNAfdMXh2c2FHdFBaQmwxTnZjbXNlbzBhY3FobXlu
bzBDTmF5SVAtX1UifV0sIm12IjoiQXhZOERDdERhR2xzYkdsamIzUm9aUSIsImNpcGhl
cnRleHQiOiJwc2o2dGtyaGJXM0lmVElMeE9GMU5HdFUtcTFmeVBidV9KwK9jbklYcWIw
eTNPOHN6OTItaWpWR1ZyRW5WbGlsY1FYeWFnZTNyX1JGdEkwV3B4UmRodyIsInRhZyI6
Ik0zb2dnNk11MVJYMUMybEZvaG5rTkN5b25qNjd2TDNqd2RrZXhFdUlpaTg1fX19",
    "protected": "eyJhbGciOiJSUzI1NiJ9",
    "header": {
      "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d",
      "signer": "
MIIC3zCCAkigAwIBAgIJAJf2ffkE1BYOMA0GCSqGSIB3DQEBBQUAMF0xChAJBgNVBA
YTA1VTMRMwEQYDVQIDApDyWxpZm9ybmlhMRMwEQYDVQQHDApDyWxpZm9ybmlhMSEw
HwYDVQQKDBhJbnRlcmlkCBXaWRnaXRzIFB0eSBMdGQwHhcNMTUwNzAyMDkwMTE4WWh
cNMjAwNjMwMDkwMTE4WjBaMQswCQYDVQQGEWJVUzETMBEGA1UECAwKQ2FsaWZvcm5p
YTETMBEGA1UEBwwKQ2FsaWZvcm5pYTEhMB8GA1UECgwYSW50ZXJuZXQgV2lkZ2l0cy
BQdHkgTHRkMIGfMA0GCSqGSIB3DQEBQUAA4GNADCBiQKBgQC8Ztxm1bYickpgSVG-
meHInI3f_chlMBdL8l7daOEztSs_a6GLqmvSu-
AoDpTsfEd4EazdMBp5fmgLRGdCYMcI6bgpO94h5CCnlj8xFKPq7qGixdwGUA6b_ZI3
c4cZ8eu73VMNrrn_z3WTZlExlpT9XVj-
ivhfJ4a6T20EtMM5qwIDAQAB04GsMIGpMHQGA1UdIwRtMGUhxQRCMF0xChAJBgNVBA
YTA1VTMRMwEQYDVQIDApDyWxpZm9ybmlhMRMwEQYDVQQHDApDyWxpZm9ybmlhMSEw
HwYDVQQKDBhJbnRlcmlkCBXaWRnaXRzIFB0eSBMdGSCCQX9nxZBNQWDjAJBgNVHR
MEAjAAMA4GA1UdDwEB_wQEAWIGWDAWBgNVHSUBAf8EDDAKBggrBgEFBQcDAzANBgkq
hkiG9w0BAQUFAAOBgQAGkz9QpoxghZUWT4ivem4cIckfxzTBBiPHCjrrjB2X8Ktn8G
SZlMdyIZV8fwdEmD90IvtMHgtzK-
9wo6Aibj_rVipxGb7trP82uzc2X8VwYnQbuqQyzofQvcwZHLyplvi95pZ5fVrJvnYA
UBFyfrdT5GjqLlnqH3a_Y3QPscuCjg"
    },
    "signature": "
Twajmt_BBLIMcNrDsJqr8lI707lEQxXZNhlUOtFkOMMqf37wOPKtp_99LoS82CVmdpCo
PLaws8zzh-SNIQ42-
9GYO8_9BaEGCiCwy18YgWP9fWNfNv2gR2f12DK4uknkYu1EMBW4YfP81n_pGpb4Gm-
nMk14grVZygaPej3ZZk"
  }
}

```

A.2.3. Sample DeleteTA

A.2.3.1. Sample DeleteTAResponse

```
{
  "DeleteTATBSRequest": {
    "ver": "1.0",
    "rid": "req-2",
    "tid": "tran-01",
    "tee": "SecuriTEE",
    "nextdsi": "false",
    "dsihash": "gwjul_9MZks3pqUSN1-eL1aViwGXNAxk0AIKW79dn4U",
    "content": {
      "tamid": "TAM1.acme.com",
      "sdname": "sd.bank.com",
      "taid": "sd.bank.com.ta"
    }
  }
}
```

```
{
  "DeleteTARrequest": {
    "payload":
      "eyJEZWxldGVUQVRCU1JlcXVlc3QiOnsidmVyIjoimS4wIiwicmlkIjoicmVxLTIiLCJOaWQiOiJ0cmFuLTAxIiwidGVlIjoieU2VjdXJpVEVFIiwibmV4dGRzaSI6ImZhbnHNlIiwiaZHNpaGFzaCI6Imd3anVsXzlnNWmtzM3BxVVNOMS1lTDfhVml3RlhOQXhrMEFJS1c3OWRuNFUiLCJjb250ZW50Ijpw7InByb3RlY3RlZCI6eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0sInJlY2lwYWVudHMlOlt7ImhlYWRLciI6eyJhbGciOiJSU0ExXzUifSwiZW5jcmlwdGVkX2tleSI6ImtyaGs0d2dpY0RlX3d0VXQyTW4tSUJsdUtvcX0JkeXpNY2plcvlBenBPYnRSTG9MZZzQ0QkfLN2tRVWE1YTg0TEVJRGEzaHNTWDIXdlldnZFJLczN4MTJsOUh5VFdfLUNSWmZtcUx2bEhlLV9MSVdvc1ZyRTZVMlJqUnRndllVOWliUkVLczkzRDRHWm4xVHFuZG9nd0tXRf9jdGlnWG1sbzZzZXpCWDZhrldZMCJ9XSwiaXYiOiJBefK4RENORGfHbHNiR2xqYjNSblpRIiwiY2lwagVydGV4dCI6IkhhNzBVdFlUS1ZCaldeUm4yLTBIxlBGal9yQnpQdGJHdzhsNktlMXotdklNeFBsyONxaIpuzmwytjrjUTZPSTZCSHZJUUFoM2Jic0l0dh1RbxhDTE5Nb8wejbBrYm9TdkiYvXlxWexpeGVZIiwidGFniJoideEtUbFRldlR2LTRtVVIgyYldyWnZMMVlhQnRGNloxVlNxoTMzmVmI2UEpmcyJ9fx0",
    "protected" : "eyJhbGciOiJSUzI1NiJ9",
    "header": {
      "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d",
      "signer": "MIIC3zCCAkgAwIBAgIJAJf2ffkE1BYOMA0GCSqGSIB3DQEBBQUAMFoxCzAJBgNVBAYTA1VTMRMwEQYDVQQIDApDYWxpZm9ybmlhMRMwEQYDVQQHDApDYWxpZm9ybmlhMSEwHwYDVQQKBhJbnRlcm5ldCBXaWRnaHRzIFB0eSBMdGQwHhcNMtUwNzAyMDkwMTE4WhcNMjAwNjMwMDkwMTE4WjBaMQswCQYDVQQGEWJVUzETMBEGA1UECAwkQ2FsaWZvcms5pYtETMBEGA1UEBwwKQ2FsaWZvcms5pYtEHBM8GA1UECgwYSW50ZXJuZXQvZ2lkZ2l0cyBqdHkgTHRkMIGfMA0GCSqGSIB3DQEBAQUAA4GNADCBiQKBggQC8ZtxM1byickpgSVGmeHIInI3f_chlMBdL8l7daOEztSs_a6GLqmvSuAoDPtsfEd4EazdMBp5fmglRGdCYMcI6bgp094h5CCnlj8xFkpq7qGixdwGUA6b_ZI3c4cZ8eu73VMNrrn_z3WTZlExlpT9Xvj-ivhfJ4a6T20EtMM5qwIDAQABo4GsMIGPMHQGA1UdIwRtMGUHxQRCMFoxCzAJBgNVBAYTA1VTMRMwEQYDVQQIDApDYWxpZm9ybmlhMRMwEQYDVQQHDApDYWxpZm9ybmlhMSEwHwYDVQQKBhJbnRlcm5ldCBXaWRnaHRzIFB0eSBMdGSCCX9nxZBNQWDjAJBgNVHRMEAIAAMA4GA1UdDwEB_wQEAWIGDAWBgnVHSubAF8EDDAKBggrBgEFBQcDAzANBgkqhkiG9w0BAQUFAAOBgQAQgz9QpoxghZUWT4ivem4cIckfxzTBBiPHCjrrjB2X8Ktn8GSZlMdyIZV8fwdEmD90IvtMHgtzK-9wo6Aibj_rVIPxGb7trP82uzc2X8VwYnQbuqQyzofQvcwZHLyplvi95pz5fvrJvnYAUBFyfrdt5GjjLlnqh3a_Y3QPscuCjg"
    },
    "signature" :
      "BZS0_Ab6pqvGNXe5lqt4Sc3jakyWQeiK9KlVSnimwWnjCCyMtyB9bwvlbILZba3IJIfe_3F9biIQpSyTGSoft2QRPTKC7psjwDw-3kh7HkHcPPJd-PpMMfQvRx7AIV8vbQo9MiJIC62iN0V2se5z2v8VFjGSORGGq225w7FvrnWE"
  }
}
```

A.2.3.2. Sample DeleteTAResponse

```
{
  "DeleteTATBSResponse": {
    "ver": "1.0",
    "status": "pass",
    "rid": "req-2",
    "tid": "tran-01",
    "content": {
      "did": "zAHkb0-SQh9U_OT8mR5dB-tygcqpUJ9_x07pIiw8WoM"
    }
  }
}
```



```
{
  "DeleteTAResponse": {
    "payload": "
ew0KCSJEZwXldGVUQVRCU1Jlc3BvbNlIjogew0KCQkidmVyIjogIjEuMCIsDQoJCSJz
dGF0dXMlOiAicGFzcyIsDQoJCSJyaWQiOiAicmVxLTlIiLA0KCQkidGlKIjogInRyYW4t
MDEiLA0KCQkiY29udGVudCI6IHsNCgkJCSJwcm90ZWN0ZWQlOnsiZW5jIjoiTTEyOENC
QylIUzI1NiJ9LA0KCQkIjInJlY2lwaWVudHMlOlslNCgkJCQl7DQoJCQkJCSJoZWfkZXIi
OnsiYWxnIjoilNBMV81In0sDQoJCQkJCSJlbnNyeXB0ZWRFa2V5IjoITXdtU1ZHaWU2
eHpfQmxTaEFlMTFRKRHhKT3oyNWVhYy1HZ2NEM2o5OWFyM2E4X21YY182ZE44bFRTb1dD
X19wZEFHaEMyWk5SakdIcTBCZ2JDYTRKalk0eXRkMVBVWDB6M1psbXl1YnRXM29leEpY
el9PMzg1WGM4S3hySndjbElyZGx2WUY2OVZmeERLQkVzUHJCdzlVenVla1VmSU4xWlFU
bWZ0QmVaSlJnIg0KCQkJCX0NCgkJCVC0sDQoJCQkiaXYiOiJBeFk4REN0RGFhbnHNiR2xq
YjNSblpRIiwNCgkJCSJjaXB0ZXJ0ZXh0IjoiamhQTlV5ZkFtel9rVV9GbEM2LUtCME0l
WDBHNE5MbHc0LWt0bERYajZTWlUteUp6eUFUbC1oY0ZBWWMWLXJMVVEF4cF93N1dlWER0
Y3N3SsZJSSzRjcWciLA0KCQkIjInRnZyI6I1lBBeGo5N25oT29qVTNIREhxSl14MGZMNWpt
b0xkTlJkTHRTAMiZUTdrYXciDQoJCX0NCgl9DQp9",
    "protected": "eyJhbGciOiJSUzI1NiJ9",
    "header": {
      "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d",
      "signer": "
MIIC3zCCAkigAwIBAgIJAJf2fFkE1BYOMA0GCSqGSIB3DQEBBQUAMF0xCzAJ
BgNVBAYTAlVTMRMwEQYDVQQIDApDYWxpZm9ybmlhMRMwEQYDVQQHDApDYWxp
Zm9ybmlhMSEwHwYDVQQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQwHhcn
MTUwNzAyMDkwMTE4W4hcnMjAwNjMwMDkwMTE4W4jBAMQswCQYDVQQGEwJVUzET
MBEGA1UECAwKQ2FsaWZvcn5pYtETMBEGA1UEBwwKQ2FsaWZvcn5pYtEhMB8G
A1UECgwYSW50ZXJuZXQgV2lkZ2l0cyBQdHkgTHRkMIGfMA0GCSqGSIB3DQEB
AQUAAAGNADCBiQKBgQC8ZtxM1bYickpgSVG-
meHInI3f_chlMBdL8l7daOEztSs_a6GLqmvSu-
AoDPtsfEd4EazdMBp5fmgLRGdCYMcI6bgp094h5CCnlj8xFKPq7qGixdwGUA
6b_ZI3c4cZ8eu73VMNrrn_z3WTZlExlpT9XVj-
ivhfJ4a6T20EtMM5qwIDAQABo4GsMIGpMHQGA1UdIwRtMGUhxQRCMF0xCzAJ
BgNVBAYTAlVTMRMwEQYDVQQIDApDYWxpZm9ybmlhMRMwEQYDVQQHDApDYWxp
Zm9ybmlhMSEwHwYDVQQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGSCCQCX
9nxZBNQWDjAJBgNVHRMEAjaAAMA4GA1UdDwEB_wQEAWIGwDAWBgNVHSUBA8f8E
DDAKBggrBgEFBQcDAzANBgkqhkiG9w0BAQUFAAOBgQAGkz9QpoxghZUWT4iv
em4cIckfxzTBBiPHCjrrjB2X8Ktn8GSZ1MdyIZV8fwdEmD90IvtMHgtzK-
9wo6Aibj_rVIpXGb7trP82uzc2X8VwYnQbuqQyzofQvcwZHLyplvi95pZ5fV
rJvnYAUBFyfrdT5GjjqLlnqh3a_Y3QPscuCjg"
    },
    "signature": "
DfoBOetNelKsnAe_m4Z9K5UbihgWNYZsp5jVybiI05sOagDzv6R4do9npaAlAvpNK8HJ
Cx6D22J8GDUExlIhSR1aDuDCQm6QzmjdfdxAz5TRYl6zPcZqgStoN_g1TZxqxEv6V
Ob5fies4g6MHvCH-I1_-KbHq5YpwGxEEfdg"
  }
}
```

A.3. Example OTrP Broker Option

The most popular TEE devices today are Android powered devices. In an Android device, an OTrP Broker can be a bound service with a service registration ID that a Client Application can use. This option allows a Client Application not to depend on any OTrP Broker SDK or provider.

An OTrP Broker is responsible to detect and work with more than one TEE if a device has more than one. In this version, there is only one active TEE such that an OTrP Broker only needs to handle the active TEE.

Appendix B. Contributors

- Brian Witten
Symantec
brian_witten@symantec.com
- Tyler Kim
Solacia
tylerkim@iotrust.kr

Authors' Addresses

Mingliang Pei
Symantec
350 Ellis St
Mountain View, CA 94043
USA

Email: mingliang_pei@symantec.com

Andrew Atyeo
Intercede
St. Mary's Road, Lutterworth
Leicestershire, LE17 4PS
Great Britain

Email: andrew.atyeo@intercede.com

Nick Cook
ARM Ltd.
110 Fulbourn Rd
Cambridge, CB1 9NJ
Great Britain

Email: nicholas.cook@arm.com

Minho Yoo
IoTrust
Suite 501, Gasanbusiness Center,165, Gasan digital1-ro
Seoul 08503
Korea

Email: minho.yoo@iotrust.kr

Hannes Tschofenig
ARM Ltd.
110 Fulbourn Rd
Cambridge, CB1 9NJ
Great Britain

Email: hannes.tschofenig@arm.com

TEEP WG
Internet-Draft
Intended status: Standards Track
Expires: 1 September 2022

D. Thaler
Microsoft
28 February 2022

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

extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	4
3. TEEP Broker	4
3.1. Use of Abstract APIs	4
4. Use of HTTP as a Transport	5
5. TEEP/HTTP Client Behavior	6
5.1. Receiving a request to install a new Trusted Application	6
5.1.1. Session Creation	7
5.2. Receiving a notification that a Trusted Application is no longer needed	7
5.3. Getting a TAM URI and message buffer back from a TEEP Agent	8
5.4. Receiving an HTTP response	8
5.5. Handling checks for policy changes	9
5.6. Error handling	9
6. TEEP/HTTP Server Behavior	10
6.1. Receiving an HTTP POST request	10
6.2. Getting an empty buffer back from the TAM	10
6.3. Getting a message buffer from the TAM	10
6.4. Error handling	10
7. Sample message flow	10
8. Security Considerations	12
9. IANA Considerations	12
10. References	12
10.1. Normative References	13
10.2. Informative References	13
Author's Address	14

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.

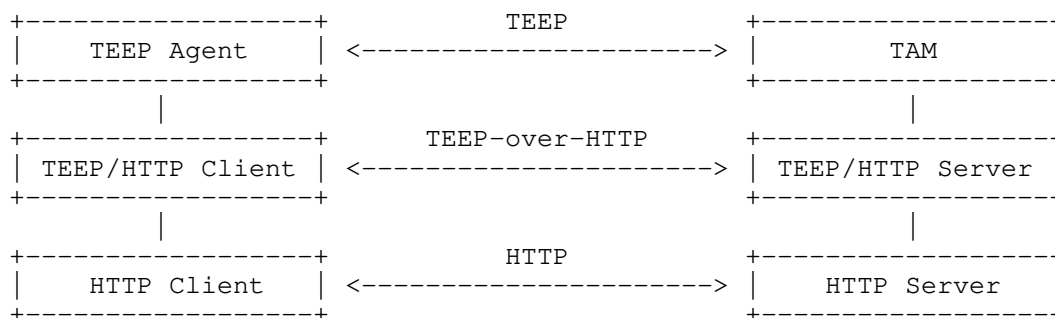


Figure 1: Agent Initiated Communication

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

- [BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<https://www.rfc-editor.org/info/rfc7525>>.
- [I-D.ietf-httpbis-semantics] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-19, 12 September 2021, <<https://www.ietf.org/archive/id/draft-ietf-httpbis-semantics-19.txt>>.
- [I-D.ietf-teep-protocol] Tschofenig, H., Pei, M., Wheeler, D., Thaler, D., and A. Tsukamoto, "Trusted Execution Environment Provisioning (TEEP) Protocol", Work in Progress, Internet-Draft, draft-ietf-teep-protocol-07, 25 October 2021, <<https://www.ietf.org/archive/id/draft-ietf-teep-protocol-07.txt>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC7925] Tschofenig, H., Ed. and T. Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things", RFC 7925, DOI 10.17487/RFC7925, July 2016, <<https://www.rfc-editor.org/info/rfc7925>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

10.2. Informative References

- [GP-OTrP] Global Platform, "TEE Management Framework: Open Trust Protocol (OTrP) Profile Version 1.0", Global Platform GPD_SPE_123, May 2019, <<https://globalplatform.org/specs-library/tee-management-framework-open-trust-protocol/>>.

- [I-D.ietf-httpbis-bcp56bis]
Nottingham, M., "Building Protocols with HTTP", Work in Progress, Internet-Draft, draft-ietf-httpbis-bcp56bis-15, 27 August 2021, <<https://www.ietf.org/archive/id/draft-ietf-httpbis-bcp56bis-15.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-architecture-15, 12 July 2021, <<https://www.ietf.org/archive/id/draft-ietf-teep-architecture-15.txt>>.
- [I-D.ietf-teep-opentrustprotocol]
Pei, M., Atyeo, A., Cook, N., Yoo, M., and H. Tschofenig, "The Open Trust Protocol (OTrP)", Work in Progress, Internet-Draft, draft-ietf-teep-opentrustprotocol-03, 15 May 2019, <<https://www.ietf.org/archive/id/draft-ietf-teep-opentrustprotocol-03.txt>>.
- [RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/info/rfc9000>>.

Author's Address

Dave Thaler
Microsoft

Email: dthaler@microsoft.com

TEEP WG
Internet-Draft
Intended status: Informational
Expires: September 6, 2019

D. Thaler
Microsoft
March 05, 2019

HTTP Transport for the Open Trust Protocol (OTrP)
draft-thaler-teep-otrp-over-http-01

Abstract

This document specifies the HTTP transport for the Open Trust Protocol (OTrP), which is used to manage code and configuration data in a Trusted Execution Environment (TEE). An implementation of this document can run outside of any TEE, but interacts with an OTrP 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 <http://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 September 6, 2019.

Copyright Notice

Copyright (c) 2019 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 (<http://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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	3
3. Use of Abstract APIs	3
4. Use of HTTP as a Transport	3
5. Client Broker Behavior	4
5.1. Receiving a request to install a new Trusted Application	4
5.1.1. Session Creation	5
5.2. Getting a message buffer back from an OTrP Agent	5
5.3. Receiving an HTTP response	6
5.4. Handling checks for policy changes	6
5.5. Error handling	7
6. Server Broker Behavior	7
6.1. Receiving an HTTP POST request	7
6.2. Getting an empty buffer back from the TAM	7
6.3. Getting a message buffer from the TAM	7
6.4. Error handling	7
7. Sample message flow	7
8. Security Considerations	9
9. IANA Considerations	10
10. References	11
10.1. Normative References	11
10.2. Informative References	11
Author's Address	12

1. Introduction

Trusted Execution Environments (TEEs), including Intel SGX, ARM TrustZone, Secure Elements, and others, enforce that only authorized code can execute within the TEE, and any memory used by such code is protected against tampering or disclosure outside the TEE. The Open Trust Protocol (OTrP) is designed to provision authorized code and configuration into TEEs.

To be secure against malware, an OTrP implementation (referred to as an OTrP "Agent" on the client side, and a "Trusted Application Manager (TAM)" on the server side) must themselves run inside a TEE. However, the transport for OTrP, along with typical networking stacks, need not 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) that only sees encrypted messages to be kept out of the TEE.

The OTrP specification [I-D.ietf-teep-opentrustprotocol] describes the behavior of OTrP Agents and TAMs, but does not specify the details of the transport, an implementation of which is referred to as a "Broker". The purpose of this document is to provide such

details. That is, the HTTP transport for OTrP is implemented in a Broker (typically outside a TEE) that delivers messages up to an OTrP implementation, and accepts messages from the OTrP implementation to be sent over a network.

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), Agent, and Broker.

3. Use of Abstract APIs

This document refers to various APIs between a Broker and an OTrP 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.

It is common in some TEE architectures (e.g., SGX) to refer to calls into a Trusted Application (TA) as "ECALLs" (or enclave-calls), and calls out from a Trusted Application (TA) as "OCALLs" (or out-calls).

In other TEE architectures, there may be no OCALLs, but merely data returned from calls into a TA. This document attempts to be agnostic as to the concrete API architecture. As such, abstract APIs used in this document will refer to calls into a TA as API calls, and will simply refer to "passing data" back out of the TA. A concrete API might pass data back via an OCALL or via data returned from an API call.

This document will also refer to passing "no" data back out of a TA. In an OCALL-based architecture, this might be implemented by not making any such call. In a return-based architecture, this might be implemented by returning 0 bytes.

4. Use of HTTP as a Transport

This document uses HTTP [I-D.ietf-httpbis-semantics] as a transport. When not called out explicitly in this document, all implementation recommendations in [I-D.ietf-httpbis-bcp56bis] apply to use of HTTP by OTrP.

Redirects MAY be automatically followed, and no additional request headers beyond those specified by HTTP need be modified or removed upon a following such a redirect.

Content is not intended to be treated as active by browsers and so HTTP responses with content SHOULD have the following headers as explained in Section 4.12 of [I-D.ietf-httpbis-bcp56bis]:

```
Content-Type: application/otrp+json
Cache-Control: no-store
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 an OTrP Agent via an out-of-band mechanism, as discussed in the next section.

When HTTPS is used, TLS certificates MUST be checked according to [RFC2818].

5. Client Broker Behavior

5.1. Receiving a request to install a new Trusted Application

When the Broker receives a notification (e.g., from an application installer) that an application has a dependency on a given Trusted Application (TA) being available in a given type of TEE, the notification will contain the following:

- A unique identifier of the TA
- Optionally, any metadata to pass to the OTrP 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 Broker.

When such a notification is received, the Broker 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.

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

The OTrP 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 broker, depending on the OTrP Agent's configuration. If they differ, the Broker MUST use the TAM URI passed back.

5.1.1. Session Creation

If no data is passed back, the Broker simply informs its client (e.g., the application installer) of success.

If the OTrP Agent passes back a TAM URI with no message buffer, the TEEP Broker attempts to create session state, then sends an HTTP(S) POST to the TAM URI with an "Accept: application/otrp+json" header and an empty body. The HTTP request is then associated with the Broker's session state.

If the OTrP Agent instead passes back a TAM URI with a message buffer, the TEEP Broker attempts to create session state and handles the message buffer as specified in Section 5.2.

Session state consists of:

- Any context (e.g., a handle) that identifies the API session with the OTrP Agent.
- Any context that identifies an HTTP request, if one is outstanding. Initially, none exists.

5.2. Getting a message buffer back from an OTrP Agent

When a message buffer (and TAM URI) is passed to a Broker from an OTrP Agent, the Broker MUST do the following, using the Broker's session state associated with its API call to the OTrP Agent.

The Broker sends an HTTP POST request to the TAM URI with "Accept: application/otrp+json" and "Content-Type: application/otrp+json" headers, and a body containing the OTrP message buffer provided by the OTrP Agent. The HTTP request is then associated with the Broker's session state.

5.3. Receiving an HTTP response

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

If the HTTP response body is empty, the Broker'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 Broker calls a "ProcessOTrPMessage" API (Section 6.2 of [I-D.ietf-teep-opentrustprotocol]) to pass the response body to the OTrP Agent associated with the session. The OTrP Agent will then pass no data back, or pass back a message buffer.

If no data is passed back, the Broker's task is complete, and it can delete its session state, and inform its client (e.g., the application installer) of success.

If instead the OTrP Agent passes back a message buffer, the TEEP Broker handles the message buffer as specified in Section 5.2.

5.4. Handling checks for policy changes

An implementation MUST provide a way to periodically check for OTrP policy changes. This can be done in any implementation-specific manner, such as:

A) The Broker might call into the OTrP Agent at an interval previously specified by the OTrP Agent. This approach requires that the Broker be capable of running a periodic timer.

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

C) The Broker might be informed when any attestation attempt determines that the device is out of compliance, and call into the OTrP Agent to remediate.

The Broker informs the OTrP Agent by invoking an appropriate "RequestPolicyCheck" API. The OTrP 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.

5.5. Error handling

If any local error occurs where the Broker cannot get a message buffer (empty or not) back from the OTrP Agent, the Broker deletes its session state, and informs its client (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 Broker calls the OTrP Agent's "ProcessError" API, and then deletes its session state and informs its client of a failure.

6. Server Broker Behavior

6.1. Receiving an HTTP POST request

When an HTTP POST request is received with an empty body, the Broker invokes the TAM's "ProcessConnect" API. The TAM will then pass back a (possibly empty) message buffer.

When an HTTP POST request is received with a non-empty body, the Broker calls the TAM's "ProcessOTrPMessage" API to pass it the request body. 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 Broker sends a successful (2xx) response with no body.

6.3. Getting a message buffer from the TAM

If the TAM passes back a non-empty buffer, the Broker generates a successful (2xx) response with a "Content-Type: application/otrp+json" header, and with the message buffer as the body.

6.4. Error handling

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

7. Sample message flow

1. An application installer determines (e.g., from an app manifest) that the application has a dependency on TA "X", and passes this notification to the Client Broker. The Client Broker picks an

OTrP Agent (e.g., the only one available) based on this notification.

2. The Client Broker calls the OTrP Agent's "RequestTA" API, passing TA Needed = X.
3. The OTrP Agent finds that no such TA is already installed, but that it can be obtained from a given TAM. The OTrP Agent passes the TAM URI (e.g., "https://example.com/tam") to the Client Broker. (If the OTrP Agent already had a cached TAM certificate that it trusts, it could skip to step 9 instead and generate a GetDeviceStateResponse.)
4. The Client Broker sends an HTTP POST request to the TAM URI:

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

5. The Server Broker receives the HTTP POST request, and calls the TAM's "ProcessConnect" API.
6. The TAM generates an OTrP message (typically GetDeviceStateRequest is the first message) and passes it to the Server Broker.
7. The Server Broker sends an HTTP successful response with the OTrP message in the body:

```
HTTP/1.1 200 OK
Content-Type: application/otrp+json
Content-Length: [length of OTrP message here]
Server: Bar/2.2
Cache-Control: no-store
X-Content-Type-Options: nosniff
Content-Security-Policy: default-src 'none'
Referrer-Policy: no-referrer

[OTrP message here]
```

8. The Client Broker gets the HTTP response, extracts the OTrP message and calls the OTrP Agent's "ProcessOTrPMessage" API to pass it the message.
9. The OTrP Agent processes the OTrP message, and generates an OTrP response (e.g., GetDeviceStateResponse) which it passes back to the Client Broker.
10. The Client Broker gets the OTrP message buffer and sends an HTTP POST request to the TAM URI, with the OTrP message in the body:

```
POST /tam HTTP/1.1
Host: example.com
Accept: application/otrp+json
Content-Type: application/otrp+json
Content-Length: [length of OTrP message here]
User-Agent: Foo/1.0
```

```
[OTrP message here]
```

11. The Server Broker receives the HTTP POST request, and calls the TAM's "ProcessOTrPMessage" API.
12. Steps 6-11 are then repeated until the TAM passes no data back to the Server Broker in step 6.
13. The Server Broker sends an HTTP successful response with no body:

```
HTTP/1.1 204 No Content
Server: Bar/2.2
```

14. The Client Broker deletes its session state.

8. Security Considerations

Although OTrP 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 Section 6 of [I-D.ietf-httpbis-bcp56bis]. As such, Broker implementations MUST support HTTPS. The choice of HTTP vs HTTPS at runtime is up to policy, where an administrator configures the TAM URI to be used, but it is expected that real deployments will always use HTTPS TAM URIs.

9. IANA Considerations

[[NOTE: This section should probably be moved to the OTrP spec.]]

This section requests that IANA assign the "application/otrp+json" media type.

Type name: application

Subtype name: otrap+json

Required parameters: none

Optional parameters: none

Encoding considerations: Same as encoding considerations of application/json as specified in Section 11 of [RFC7159].

Security considerations: See Section 12 of [RFC7159] and Section 8 of this document.

Interoperability considerations: Same as interoperability considerations of application/json as specified in [RFC7159].

Published specification: [I-D.ietf-teep-opentrustprotocol]

Applications that use this media type: OTrP 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

Person & email address to contact for further information:
teep@ietf.org

Intended usage: COMMON

Restrictions on usage: none

Author: See the "Authors' Addresses" section of this document.

Change controller: IETF

10. References

10.1. Normative References

- [I-D.ietf-httpbis- semantics]
Fielding, R., Nottingham, M., and J. Reschke, "HTTP Semantics", draft-ietf-httpbis- semantics-03 (work in progress), October 2018.
- [I-D.ietf-teep-opentrustprotocol]
Pei, M., Atyeo, A., Cook, N., Yoo, M., and H. Tschofenig, "The Open Trust Protocol (OTrP)", draft-ietf-teep-opentrustprotocol-02 (work in progress), October 2018.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<https://www.rfc-editor.org/info/rfc2818>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

10.2. Informative References

- [I-D.ietf-httpbis-bcp56bis]
Nottingham, M., "Building Protocols with HTTP", draft-ietf-httpbis-bcp56bis-08 (work in progress), November 2018.
- [I-D.ietf-teep-architecture]
Pei, M., Tschofenig, H., Wheeler, D., Atyeo, A., and D. Liu, "Trusted Execution Environment Provisioning (TEEP) Architecture", draft-ietf-teep-architecture-01 (work in progress), October 2018.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.

Author's Address

David Thaler
Microsoft

EMail: dthaler@microsoft.com

TEEP
Internet-Draft
Intended status: Standards Track
Expires: January 9, 2020

M. Pei
Symantec
H. Tschofenig
Arm Ltd.
D. Wheeler
Intel
July 8, 2019

The Open Trust Protocol (OTrP) v2
draft-tschofenig-teep-otrp-v2-00

Abstract

This document specifies the Open Trust Protocol (OTrP) version 2, a protocol that provisions and installs, updates, and deletes Trusted Applications in a device with a Trusted Execution Environment (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 January 9, 2020.

Copyright Notice

Copyright (c) 2019 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 Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Requirements Language	2
3. OTrP Message Overview	3
4. Detailed Messages Specification	4
5. Security Consideration	11
6. IANA Considerations	13
7. References	15
7.1. Normative References	15
7.2. Informative References	16
Appendix A. Acknowledgements	17
Appendix B. Contributors	17
Authors' Addresses	17

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 an TEE ecosystem, different device vendors may use different operating systems in the REE and may use different types of TEEs. When application providers or device administrators use Trusted Application Managers (TAMs) to install, update, and delete Trusted Applications (TAs) on a wide range of devices with potentially different TEEs then an interoperability need arises.

This document specifies version 2 of the Open Trust Protocol (OTrP), a protocol for communicating between an OTrP server (as part of a TAM) and an OTrP client (which is a client-side component running in the REE).

The Trusted Execution Environment Provisioning (TEEP) architecture document [I-D.ietf-teep-architecture] has set to provide a design guidance for such an interoperable protocol.

2. Requirements Language

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 [RFC2119].

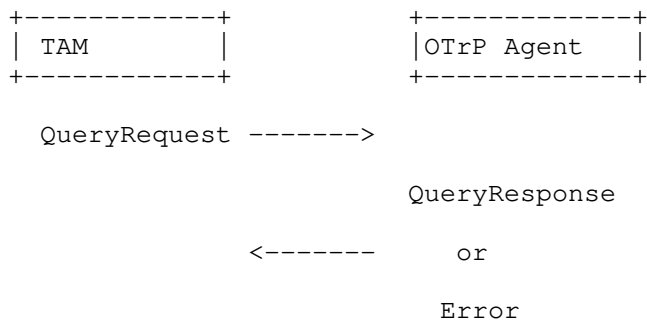
This specification re-uses the terminology defined in [I-D.ietf-teep-architecture].

3. OTrP Message Overview

OTrP consists of a couple of messages exchanged between a TAM and an OTrP Agent via an OTrP Broker. The messages are encoded either in JSON or CBOR and designed to provide end-to-end security. OTrP messages are signed and/or encrypted by the endpoints, i.e., TAM and OTrP Agent, but trusted applications may as well be encrypted and signed by the service provider. OTrP not only re-uses JSON and CBOR but also the respective security wrappers, namely JOSE (JWS [RFC7515] and JWE [RFC7516], to be more specific) and COSE [RFC8152]. Furthermore, for attestation the Entity Attestation Token (EAT) and for software updates the SUIT manifest format is re-used.

This specification defines six messages.

A TAM queries a device's current state with a QueryRequest message. An OTrP Agent will, after authenticating and authorizing the request, report attestation information, list all TAs, 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 subsequent message exchanges to install, update, or delete trusted applications shall be initiated.



With the TrustedAppInstall message a TAM can instruct an OTrP Agent to install a TA. The OTrP Agent will process the message, determine whether the TAM is authorized and whether the TA has been signed by an authorized SP. In addition to the binary, the TAM may also provide personalization data. If the TrustedAppInstall message was processed successfully then a Success message is returned to the TAM, an Error message otherwise.

TAM	OTrP Agent
-----	------------

TrustedAppInstall ---->

Success

<---- or

Error

With the TrustedAppDelete message a TAM can instruct an OTrP Agent to delete one or multiple TA(s). A Success message is returned when the operation has been completed successfully, and an Error message otherwise.

TAM	OTrP Agent
-----	------------

TrustedAppDelete ---->

Success

<---- or

Error

4. Detailed Messages Specification

For a CBOR-based encoding the following security wrapper is used (described in CDDL format [I-D.ietf-cbor-cddl]).

```

Outer_Wrapper = {
    msg-authenc-wrapper      => bstr .cbor
                              Msg_AuthEnc_Wrapper / nil,
    otrp-message             => (QueryRequest /
                              QueryResponse /
                              TrustedAppInstall /
                              TrustedAppDelete /
                              Error /
                              Success ),
}

msg-authenc-wrapper = 1
otrp-message = 2

Msg_AuthEnc_Wrapper = [ * (COSE_Mac_Tagged /
                           COSE_Sign_Tagged /
                           COSE_Mac0_Tagged /
                           COSE_Sign1_Tagged) ]

```

A future version of this specification will also describe the security wrapper for JSON (in CDDL format).

```
suite = int
```

```
version = int
```

```

data_items = (
    attestation: 1,
    ta: 2,
    ext: 3
)

```

```

QueryRequest = (
    TYPE : int,
    TOKEN : bstr,
    REQUEST : [+data_items],
    ? CIPHER_SUITE : [+suite],
    ? NONCE : bstr,
    ? VERSION : [+version],
    ? OCSP_DATA : bstr,
    * $$extensions
)

```

A QueryRequest message is signed by the TAM and has the following fields:

TYPE TYPE = 1 corresponds to a QueryRequest message sent from the TAM to the OTrP Agent.

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

REQUEST The REQUEST field indicates what information the TAM requests from the OTrP Agent in form of a list of integer values. Each integer value corresponds to an IANA registered information element. This specification defines the initial set of information elements. With 'attestation' (1) the TAM requests the OTrP Agent to return an EAT entity attestation token in the response, with 'ta' (2) the TAM wants to query the OTrP Agent for all installed TAs, and with 'ext' (3) the TAM wants to query the OTrP Agent for supported extensions. Further values may be added in the future via IANA registration.

CIPHER_SUITE The CIPHER_SUITE field lists the ciphersuite(s) supported by the TAM.

NONCE NONCE is an optional field used for ensuring the refreshness of the Entity Attestation Token (EAT) contained in the response.

VERSION The VERSION field lists the version(s) supported by the TAM. For this version of the specification this field can be omitted.

OCSP_DATA The OCSP_DATA field contains a list of OCSP stapling data respectively for the TAM certificate and each of the CA certificates up to the root certificate. The TAM provides OCSP data so that the OTrP Agent can validate the status of the TAM certificate chain without making its own external OCSP service call.

```
ta_id = (  
    Vendor_ID = bstr,  
    Class_ID = bstr,  
    Device_ID = bstr,  
    * $$extensions  
)  
  
ext_info = int  
  
QueryResponse = (  
    TYPE : int,  
    TOKEN : bstr,  
    ? SELECTED_CIPHER_SUITE : suite,  
    ? SELECTED_VERSION : version,  
    ? EAT : bstr,  
    ? TA_LIST : [+ta_id],  
    ? EXT_LIST : [+ext_info],  
    * $$extensions  
)
```

The QueryResponse message is signed and encrypted by the OTrP Agent and returned to the TAM. It has the following fields:

TYPE TYPE = 2 corresponds to a QueryResponse message sent from the OTrP Agent to the TAM.

TOKEN The value in the TOKEN field is used to match requests to responses. The value MUST correspond to the value received with the QueryRequest.

SELECTED_CIPHER_SUITE The SELECTED_CIPHER_SUITE field indicates the selected ciphersuite.

SELECTED_VERSION The SELECTED_VERSION field indicates the OTrP protocol version selected by the OTrP Agent.

EAT The EAT field contains an Entity Attestation Token following the encoding defined in [I-D.ietf-rats-eat].

TA_LIST The TA_LIST field enumerates the trusted applications installed on the device in form of ta_ids, i.e., a vendor id/class id/device id triple.

EXT_LIST The EXT_LIST field lists the supported extensions. This document does not define any extensions.

```
TrustedAppInstall = (  
    TYPE : int,  
    TOKEN : bstr,  
    ? TA  : [+SUIT_Outer_Wrapper],  
    * $$extensions  
)
```

The TrustedAppInstall message is MACed and encrypted by the TAM and has the following fields:

TYPE TYPE = 3 corresponds to a TrustedAppInstall message sent from the TAM to the OTrP Agent. In case of successful processing, an Success message is returned by the OTrP Agent. In case of an error, an Error message is returned. Note that the TrustedAppInstall message is used for initial TA installation but also for TA updates.

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

TA The TA field is used to convey one or multiple SUIT manifests. The SUIT manifest contains the code for the trusted app but may also convey personalization data. TA binaries and personalization data is often signed and encrypted by the SP. 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 SP sign and/or encrypt the TA binary.

```
TrustedAppDelete = (  
    TYPE : int,  
    TOKEN : bstr,  
    ? TA_LIST : [+ta_id],  
    * $$extensions  
)
```

The TrustedAppDelete message is MACed and encrypted by the TAM and has the following fields:

TYPE TYPE = 4 corresponds to a TrustedAppDelete message sent from the TAM to the OTrP Agent. In case of successful processing, an Success message is returned by the OTrP Agent. In case of an error, an Error message is returned.

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

TA_LIST The TA_LIST field enumerates the TAs to be deleted.

```
Success = (  
    TYPE : int,  
    TOKEN : bstr,  
    ? MSG : tstr,  
    * $$extensions  
)
```

The Success message is MACed and encrypted by the OTrP Agent and has the following fields:

TYPE TYPE = 5 corresponds to a Error message sent from the OTrP Agent to the TAM.

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

MSG The MSG field contains optional diagnostics information encoded in UTF-8 [RFC3629] returned by the OTrP Agent.

```
Error = (  
    TYPE : int,  
    TOKEN : bstr,  
    ERR_CODE : int,  
    ? ERR_MSG : tstr,  
    ? CIPHER_SUITE : [+suite],  
    ? VERSION : [+version],  
    * $$extensions  
)
```

If possible, the Error message is MACed and encrypted by the OTrP Agent. Unprotected Error messages MUST be handled with care by the TAM due to possible downgrading attacks. It has the following fields:

TYPE TYPE = 6 corresponds to a Error message sent from the OTrP Agent to the TAM.

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

ERR_CODE The ERR_CODE field is populated with values listed in a registry (with the initial set of error codes listed below). Only selected messages are applicable to each message.

ERR_MSG The ERR_MSG message is a human-readable diagnostic message that MUST be encoded using UTF-8 [RFC3629] using Net-Unicode form [RFC5198].

VERSION The VERSION field enumerates the protocol version(s) supported by the OTrP Agent. This field is optional but MUST be returned with the ERR_UNSUPPORTED_MSG_VERSION error message.

CIPHER_SUITE The CIPHER_SUITE field lists the ciphersuite(s) supported by the OTrP Agent. This field is optional but MUST be returned with the ERR_UNSUPPORTED_CRYPT_ALG error message.

This specification defines the following initial error messages. Additional error code can be registered with IANA.

ERR_ILLEGAL_PARAMETER The OTrP Agent sends this error message when a request contains incorrect fields or fields that are inconsistent with other fields.

ERR_UNSUPPORTED_EXTENSION The OTrP Agent sends this error message when it recognizes an unsupported extension or unsupported message.

ERR_REQUEST_SIGNATURE_FAILED The OTrP Agent sends this error message when it fails to verify the signature of the message.

ERR_UNSUPPORTED_MSG_VERSION The OTrP Agent receives a message but does not support the indicated version.

ERR_UNSUPPORTED_CRYPT_ALG The OTrP Agent receives a request message encoded with an unsupported cryptographic algorithm.

ERR_BAD_CERTIFICATE The OTrP Agent returns this error when processing of a certificate failed. For diagnosis purposes it is RECOMMENDED to include information about the failing certificate in the error message.

ERR_UNSUPPORTED_CERTIFICATE The OTrP Agent returns this error when a certificate was of an unsupported type.

ERR_CERTIFICATE_REVOKED The OTrP Agent returns this error when a certificate was revoked by its signer.

ERR_CERTIFICATE_EXPIRED The OTrP Agent returns this error when a certificate has expired or is not currently valid.

ERR_INTERNAL_ERROR The OTrP Agent returns this error when a miscellaneous internal error occurred while processing the request.

ERR_RESOURCE_FULL This error is reported when a device resource isn't available anymore, such as storage space is full.

ERR_TA_NOT_FOUND This error will occur when the target TA does not exist. This error may happen when the TAM has stale information and tries to delete a TA that has already been deleted.

ERR_TA_ALREADY_INSTALLED While installing a TA, a TEE will return this error if the TA has already been installed.

ERR_TA_UNKNOWN_FORMAT The OTrP Agent returns this error when it does not recognize the format of the TA binary.

ERR_TA_DECRYPTION_FAILED The OTrP Agent returns this error when it fails to decrypt the TA binary.

ERR_TA_DECOMPRESSION_FAILED The OTrP Agent returns this error when it fails to decompress the TA binary.

ERR_MANIFEST_PROCESSING_FAILED The OTrP Agent returns this error when manifest processing failures occur that are less specific than **ERR_TA_UNKNOWN_FORMAT**, **ERR_TA_UNKNOWN_FORMAT**, and **ERR_TA_DECOMPRESSION_FAILED**.

ERR_PD_PROCESSING_FAILED The OTrP Agent returns this error when it fails to process the provided personalization data.

5. Security Consideration

This section summarizes the security considerations discussed in this specification:

Cryptographic Algorithms This specification relies on the cryptographic algorithms provided by the security wrappers JOSE and COSE, respectively. A companion document makes algorithm recommendations but this document is written in an algorithm-agnostic way. OTrP messages between the TAM and the OTrP Agent are protected using JWS and JWE (for JSON-encoded messages) and COSE (for CBOR-encoded messages). Public key based authentication is used to by the OTrP Agent to authenticate the TAM and vice versa.

Attestation A TAM may rely on the attestation information provided by the OTrP Agent and the Entity Attestation Token is re-used to convey this information. To sign the Entity Attestation Token it is necessary for the device to possess a public key (usually in the form of a certificate) 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 information or in the certificate used to sign the attestation token. This uniqueness may raise privacy concerns.

To lower the privacy implications the OTrP Agent MUST present its attestation information only to an authenticated and authorized TAM.

TA Binaries TA binaries are provided by the SP. It is the responsibility of the TAM to relay only verified TAs from authorized SPs. Delivery of that TA to the OTrP Agent is then the responsibility of the TAM and the OTrP Broker, using the security mechanisms provided by the OTrP. To protect the TA binary the SUIT manifest is re-used and it offers a variety of security features, including digital signatures and symmetric encryption.

Personalization Data An SP or a TAM can supply personalization data along with a TA. This data is also protected by a SUIT manifest. The personalization data may be itself is (or can be) opaque to the TAM.

OTrP Broker OTrP relies on the OTrP Broker to relay messages between the TAM and the OTrP Agent. When the OTrP Broker is compromised it can drop, relay, and replay messages but it cannot modify those messages. A compromised OTrP Broker could reorder TAM messages to install an old version of a TA. Information in the manifest ensures that the OTrP Agents are protected against such downgrading attacks.

CA Compromise The QueryRequest message from a TAM to the OTrP Agent may include OCSP stapling data for the TAM's signer certificate and for intermediate CA certificates up to the root certificate so that the OTrP Agent can verify the certificate's revocation status.

A certificate revocation status check on a TA signer certificate is OPTIONAL by an OTrP Agent. A TAM is responsible for vetting a TA and before distributing them to OTrP Agents. The OTrP Agents will trust a TA signer certificate's validation status done by a TAM.

CA Compromise The CA issuing certificates to a TAM or an SP may get compromised. A compromised intermediate CA certificates can be detected by an OTrP Agent by using OCSP information, assuming the revocation information is available. Additionally, it is RECOMMENDED to provide a way to update the trust anchor store used by the device, 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.

Compromised TAM The OTrP Agent SHOULD use OCSP information to verify the validity of the TAM-provided certificate (as well as the validity of intermediate CA certificates). The integrity and the accuracy of the clock within the TEE determines the ability to determine an expired or revoked certificate since OCSP stapling includes signature generation time, certificate validity dates are compared to the current time.

6. IANA Considerations

There are two IANA requests: a media type and list of error codes.

IANA is requested to assign a media type for application/otrpv2+json.

Type name: application

Subtype name: otrap+json

Required parameters: none

Optional parameters: none

Encoding considerations: Same as encoding considerations of application/json as specified in Section 11 of [RFC7159]

Security considerations: See Security Considerations Section of this document.

Interoperability considerations: Same as interoperability considerations of application/json as specified in [RFC7159]

Published specification: This document.

Applications that use this media type: OTrPv2 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

Person to contact for further information: teep@ietf.org

Intended usage: COMMON

Restrictions on usage: none

Author: See the "Authors' Addresses" section of this document

Change controller: IETF

IANA is requested to assign a media type for application/otrpv2+cbor.

Type name: application

Subtype name: otrpv2+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: OTrPv2 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

Person to contact for further information: teep@ietf.org

Intended usage: COMMON

Restrictions on usage: none

Author: See the "Authors' Addresses" section of this document

Change controller: IETF

IANA is also requested to create a new registry for the error codes defined in Section 4.

Registration requests are evaluated after a three-week review period on the `otrp-reg-review@ietf.org` mailing list, on the advice of one or more Designated Experts [RFC8126]. However, to allow for the allocation of values prior to publication, the Designated Experts may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register an error code: example"). Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the `iesg@ietf.org` mailing list) for resolution.

Criteria that should be applied by the Designated Experts includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or whether it is useful only for a single extension, and whether the registration description is clear.

IANA must only accept registry updates from the Designated Experts and should direct all requests for registration to the review mailing list.

7. References

7.1. Normative References

- [I-D.ietf-rats-eat] Mandyam, G., Lundblade, L., Ballesteros, M., and J. O'Donoghue, "The Entity Attestation Token (EAT)", draft-ietf-rats-eat-01 (work in progress), July 2019.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC5198] Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, DOI 10.17487/RFC5198, March 2008, <<https://www.rfc-editor.org/info/rfc5198>>.
- [RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <<https://www.rfc-editor.org/info/rfc7049>>.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.
- [RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<https://www.rfc-editor.org/info/rfc7516>>.
- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<https://www.rfc-editor.org/info/rfc7517>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<https://www.rfc-editor.org/info/rfc7518>>.
- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.

7.2. Informative References

- [I-D.ietf-cbor-cddl] Birkholz, H., Vigano, C., and C. Bormann, "Concise data definition language (CDDL): a notational convention to express CBOR and JSON data structures", draft-ietf-cbor-cddl-08 (work in progress), March 2019.

[I-D.ietf-teep-architecture]

Pei, M., Tschofenig, H., Wheeler, D., Atyeo, A., and D. Liu, "Trusted Execution Environment Provisioning (TEEP) Architecture", draft-ietf-teep-architecture-02 (work in progress), March 2019.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.

Appendix A. Acknowledgements

This work is based on the initial version of OTrP and hence credits go to those who have contributed to it.

Appendix B. Contributors

We would like to thank the following individuals for their contributions to an earlier version of this specification.

- Brian Witten
Symantec
brian_witten@symantec.com
- Tyler Kim
Solacia
tylerkim@iotrust.kr
- Nick Cook
Arm Ltd.
nicholas.cook@arm.com
- Minhoo Yoo
IoTrust
minho.yoo@iotrust.kr

Authors' Addresses

Mingliang Pei
Symantec
350 Ellis St
Mountain View, CA 94043
USA

Email: mingliang_pei@symantec.com

Hannes Tschofenig
Arm Ltd.
110 Fulbourn Rd
Cambridge, CB1 9NJ
Great Britain

Email: hannes.tschofenig@arm.com

David Wheeler
Intel
US

Email: david.m.wheeler@intel.com