GlobalPlatform Technology

TEE API Call Validation

Version 0.0.0.9

Public Review

November 2024

Document Reference: GPP_TEN_012

Contents

Tables

1 Introduction

The aim of this document is to provide a modification to the GlobalPlatform TEE Internal Core API ([TEE Core]). It is applicable to all versions of [TEE Core], and provides an OPTIONAL extension that is recommended to TEE implementors.

If you are implementing this specification and you think it is not clear on something:

1. Check with a colleague.

And if that fails:

2. Contact GlobalPlatform at TEE-issues-GPP_TEN_012@globalplatform.org

1.1 Audience

This document is intended primarily for the use of TEE implementors and Trusted Application developers.

1.2 IPR Disclaimer

Attention is drawn to the possibility that some of the elements of this GlobalPlatform specification or other work product may be the subject of intellectual property rights (IPR) held by GlobalPlatform members or others. For additional information regarding any such IPR that have been brought to the attention of GlobalPlatform, please visit https://globalplatform.org/specifications/ip-disclaimers/. GlobalPlatform shall not be held responsible for identifying any or all such IPR, and takes no position concerning the possible existence or the evidence, validity, or scope of any such IPR.

1.3 References

The table below lists references applicable to this specification. The latest version of each reference applies unless a publication date or version is explicitly stated.

Table 1-1: Normative References

Standard / Specification	Description	Ref
GPD_SPE_010	GlobalPlatform Technology TEE Internal Core API Specification (all versions)	[TEE Core]

1.4 Terminology and Definitions

Selected terms used in this document are defined in [TEE Core].

1.5 Abbreviations and Notations

Table 1-2: Abbreviations and Notations

Abbreviation / Notation	Meaning
EPFL	École Polytechnique Fédérale de Lausanne
REE	Regular Execution Environment
TA	Trusted Application
TEE	Trusted Execution Environment

1.6 Revision History

GlobalPlatform technical documents numbered *n*.0 are major releases. Those numbered *n*.1, *n*.2, etc., are minor releases where changes typically introduce supplementary items that do not impact backward compatibility or interoperability of the specifications. Those numbered *n.n*.1, *n.n*.2, etc., are maintenance releases that incorporate errata and clarifications; all non-trivial changes are indicated, often with revision marks.

Table 1-3: Revision History

Date	Version	Description
Aug 2024	0.0.0.4	Committee Review
Aug 2024	0.0.0.7	Member Review
Nov 2024	0.0.0.9	Public Review
TBD	TBD	Public Release

2 OVERVIEW

The GlobalPlatform TEE Internal Core API ([TEE Core]) describes how calls to the Trusted Application (TA) are made. In the current [TEE Core], the TA developer is responsible for validating that the caller is as expected and that parameters passed by the caller are of the appropriate types and values.

Security researchers at EPFL in Switzerland noted that many published vulnerabilities in Trusted Applications are related to TA developers failing to validate parameter types. Additional discussion at GlobalPlatform identified other areas of common concern.

This document proposes additions to [TEE Core] to allow TA developers to declare the intended usage of their APIs, and have the TEE perform validation for them. It is believed this will reduce the likelihood of future vulnerabilities. An additional advantage is that the proposed changes will simplify the code the developer has to write, aiding readability.

The proposed changes can be applied to any version of [TEE Core]. While GlobalPlatform intends to add the new capabilities into the next iteration of the API as an optional extension, the change are published here to encourage TEE vendors to adopt them early, even if their TEE provides an earlier version of the API.

3 Common Pitfalls Addressed by Proposal

- Trusted Application developers are usually responsible for validating that each parameter passed to `TA_OpenSessionEntryPoint` or `TA_InvokeCommandEntryPoint` is of the appropriate type. Failure to check is a security error that can be and has been exploited. This proposal allows a developer to declare the expected types and have the TEE validate them, avoiding risks of errors/omissions in developer validation code, and additionally avoiding the need for the developer to write this code.
- Trusted Application developers are responsible for copying parameters to (private) secure world memory in order to prevent an attack where a shared buffer is modified during operation. Similarly a TA that assembles a result to pass to a client needs to be wary of an attacker viewing or modifying it prior to completion of the call. This proposal allows a developer to indicate when the TEE should copy/isolate buffer parameters.
- Trusted Application developers may fail to validate that a buffer parameter is within a given size range. This proposal allows minimum and maximum sizes for each parameter to be specified.
- Trusted Application developers are responsible for validating who the caller is and restricting access for unexpected callers. Although in the general case this logic can be complex, this proposal provides a simple means for listing good callers and having the TEE prevent access by others.

4 API Extension

The proposed API extension defines a few new calls that may be made by a TA developer, in `TA_CreateEntryPoint`, to register policy for callers and/or commands. Calling these new APIs enables the additional validation logic.

4.1 New Types

4.1.1 TEE_ParameterPolicy

A set of one to four `TEE_ParameterPolicy` defines the policy for a single parameter.

• `type`	One of the `TEE_PARAM_TYPE_*` values listed in [TEE Core] section 4.2.1.
• `mayBeShared`	If `false`, then the TEE must isolate the buffer from the external caller. If `true`, then the buffer may or may not be shared, depending on the TEE implementation. Note that isolation must be applied for both REE and TEE callers. A simple approach would be to copy the parameter to/from TA private memory, although implementations are free to use other approaches such as manipulation of memory firewalls.
• `minSize`	Minimum buffer size for any `TEE_PARAM_TYPE_MEMREF_*` type. Ignored for other types.
• `maxSize`	Maximum buffer size for any `TEE_PARAM_TYPE_MEMREF_*` type, or `0` indicating unbounded size. Ignored for other types.

```
typedef struct {
        uint32_t   type;
        bool       mayBeShared;
        size_t     minSize;
        size_t     maxSize; // 0=don't check
} TEE_ParameterPolicy;
```

Note that `TEE_ParameterPolicy` is constructed to allow default values to be omitted in static initializers, for brevity and readability.

4.1.2 TEE_InvokeTACommandPolicy

`TEE_InvokeTACommandPolicy` indicates a valid policy for a call to `TEE_InvokeTACommand`.

- `commandId` The identifier of the command this policy relates to.
- `params[4]` A valid set of parameter policies to use with this `commandId`.

```
typedef struct {
    uint32_t              commandId;
    TEE_ParameterPolicy   params[4];
} TEE_InvokeTACommandPolicy;
```

4.1.3 TEE_OpenTASessionPolicy

`TEE_OpenTASessionPolicy` indicates a valid policy for a call to `TEE_OpenTASession`.

- `params[4]` A valid set of parameter policies to use in a call to `TEE_OpenTASession`.

```
typedef struct {
    TEE_ParameterPolicy   params[4];
} TEE_OpenTASessionPolicy;
```

4.2 New Commands

4.2.1 TEE_RegisterCommandPolicy

```
TEE_Result TEE_RegisterCommandPolicy(
    const TEE_InvokeTACommandPolicy   *policy,
    size_t                             numPolicy);
```

Description

The `TEE_RegisterCommandPolicy` function may only be called in `TA_CreateEntryPoint` or a subroutine called from it. The function registers a set of permitted command policies and modifies the behavior of any subsequent call to `TA_InvokeCommandEntryPoint`.

Note

- If the function is not called, then no checking/isolating of parameters to `TA_OpenSessionEntryPoint` takes place.
- The function may be called more than once, and all policies are applied.
- It is an error to register more than one policy for a (`commandId`, parameter type combination).
- In general, is advisable to only have one policy per `commandId`, in order to keep code simple and readable; however, multiple policies for the same `commandId` with different parameter types are permitted in order to allow legacy code to be migrated to use the new capabilities.
- If `TEE_RegisterCommandPolicy` is called one or more times, then the TEE implementation must validate all calls to `TEE_InvokeCommandEntryPoint` to ensure that the policy is met.

Parameters

• `policy`	A pointer to an array of policies. The array must not be freed until `TA_DestroyEntryPoint` is called. (Typically, the array will be statically defined.)
• `numPolicies`	The size of the policy array.

Return Value

`TEE_SUCCESS` if the policy is acceptable.

In other cases, the effect of the call is a 'no op' with a returned error code indicating the issue:

- `TEE_ERROR_BAD_PARAMETERS` if the policy conflicts with one previously registered, or if two or more entries within the policy conflict with each other.
- `TEE_ERROR_OUT_OF_MEMORY` if the policy cannot be stored.

4.2.2 TEE_RegisterOpenSessionEntryPointPolicy

```
TEE_Result TEE_RegisterOpenSessionEntryPointPolicy(
    const TEE_OpenTASessionPolicy   *policy,
    size_t                           numPolicy);
```

Description

The `TEE_RegisterOpenSessionEntryPointPolicy` function may only be called in `TA_CreateEntryPoint` or a subroutine called from it. The function registers a set of permitted policies for the open session operation and modifies the behavior of any subsequent call to `TA_OpenSessionEntryPoint`.

Note

- If the function is not called, then no checking/isolating of parameters to `TA_OpenSessionEntryPoint` takes place.
- The function may be called only once. (An exception being if all previous calls returned an error.)
- It is an error to register more than one policy for each set of parameter types.
- If `TEE_RegisterOpenSessionEntryPointPolicy` is called, then the TEE implementation must validate all calls to `TA_OpenSessionEntryPoint` to ensure that the policy is met.

Parameters

• `policy`	A pointer to an array of policies. The array must not be freed until `TA_DestroyEntryPoint` is called. (Typically, the array will be statically defined.)
• `numPolicies`	The size of the policy array.

Return Value

`TEE_SUCCESS` if the policy is acceptable.

In other cases, the effect of the call is a ‘no op’ with a returned error code indicating the issue:

- `TEE_ERROR_BAD_PARAMETERS` if the function has already been called, or if two or more entries within the policy conflict with each other.
- `TEE_ERROR_OUT_OF_MEMORY` if the policy cannot be stored.

4.2.3 TEE_RegisterPermittedCaller

```
TEE_Result TEE_RegisterPermittedCaller(const TEE_Identity *caller);
```

Description

The `TEE_RegisterPermittedCaller` function may only be called in `TA_CreateEntryPoint` or a subroutine called from it. The function registers the identity of a caller permitted to create a session to the TA, and modifies the behavior of any subsequent call to `TA_OpenSessionEntryPoint`.

Note

- If the function is not called, then no checking of the caller is made. In this circumstance a TEE implementation may choose to log a warning each time a client is about to call `TA_OpenSessionEntryPoint`, logging the identity of the caller and indicating that no access policy is in place.
- The function may be called multiple times to register multiple permitted callers.
- If `TEE_RegisterPermittedCaller` is called, then the TEE implementation must validate all calls to `TA_OpenSessionEntryPoint` to ensure that the caller has been registered.
- Additional registration methods may be added in future to identify clients by other means.

Parameters

• `caller`	A pointer to a permitted client identity. This identity structure must not be freed until `TA_DestroyEntryPoint` is called. (Typically, the identity structure will be statically defined.)

Return Value

`TEE_SUCCESS` if the caller is successfully added to the list of permitted callers.

`TEE_ERROR_OUT_OF_MEMORY` if the caller identity cannot be stored.

4.3 Impact on Existing APIs

4.3.1 TA_OpenSessionEntryPoint

Impact

If a call to `TEE_RegisterPermittedCaller` was made and the caller is NOT one of the permitted callers, then the TEE will return `TEE_ERROR_ACCESS_DENIED` with the call origin `TEE_ORIGIN_TEE`.

If a call to `TEE_RegisterOpenSessionEntryPointPolicy` was made, then:

- If the caller-supplied parameters match a policy included in the list of registered policies, then:
    - If the TEE is able to enforce that policy (e.g. allocate and copy memory), then the policy is enforced and the call proceeds.
    - If the TEE is unable able to enforce that policy (e.g. allocate and copy memory), then an appropriate error (for example `TEE_ERROR_OUT_OF_MEMORY`) is returned with call origin `TEE_ORIGIN_TEE`.
- If the caller-supplied parameters do not match any of the registered policies, then the TEE will return `TEE_ERROR_BAD_PARAMETERS` with the call origin `TEE_ORIGIN_TEE`.

4.3.2 TA_InvokeCommandEntryPoint

Impact

If no call to `TEE_RegisterCommandPolicy` was made, then the operation of this function is not modified.

If the caller supplies a combination of command id and parameters that matches a registered policy, then:

- If the TEE is able to enforce that policy (e.g. allocate and copy memory), then the policy is enforced and the call proceeds.
- If the TEE is unable to meet the policy, then an appropriate error (typically `TEE_ERROR_OUT_OF_MEMORY`) is returned with the call origin `TEE_ORIGIN_TEE`.

If the caller supplies a combination of command id and parameters that does not match a registered policy, then `TEE_ERROR_BAD_PARAMETERS` is returned with the call origin `TEE_ORIGIN_TEE`.

4.4 Simplification of User TA Code

The proposed changes are backward compatible. Existing user code will not be invalidated by application of these changes; however, there is scope for simplification of user code.

- When `TEE_RegisterCommandPolicy` has been used
  - `TEE_InvokeCommandEntryPoint` does not need to check the types of parameters for a given `commandId`, unless more than one permitted set of parameter types was registered for that `commandId` (which is supported but not recommended).
  - `TEE_InvokeCommandEntryPoint` does not need to copy IN/INOUT parameters to TEE memory, or copy INOUT/OUT parameters from TEE memory *unless* the parameter was marked as `mayBeShared` in the policy in force.
  - Note that it remains the responsibility of the developer to validate the content of any buffer.

- When `TEE_RegisterOpenSessionEntryPointPolicy` has been used
  - `TA_OpenSessionEntryPoint` does not need to check the types of parameters unless more than one permitted set of parameter types was registered.
  - `TA_OpenSessionEntryPoint` does not need to copy IN/INOUT parameters to TEE memory, or copy INOUT/OUT parameters from TEE memory *unless* the parameter was marked as `mayBeShared` in the policy in force.
  - Note that it remains the responsibility of the developer to validate the content of any buffer.

- When `TEE_RegisterPermittedCaller` has been used
  - `TA_OpenSessionEntryPoint and TEE_InvokeCommandEntryPoint` do not need to validate the caller for simple access control case.
  - If the TA wishes to restrict calls based on data passed, or other contextual information, it remains the responsibility of user code to make that determination.

4.5 Example

A portion of an example 'Hello World' TA:

```
static const TEE_InvokeTACommandPolicy policy[] = {
    {
        .commandId = CMD_SAY_HELLO,
        .params = {
            {
                .type = TEE_PARAM_TYPE_MEMREF_INPUT,
                .minSize = 1,
                .maxSize = 1024
                // Note mayBeShared=false is a default value so may be omitted
            },{
                .type = TEE_PARAM_TYPE_MEMREF_OUTPUT,
                .maxSize = 1024,
                .mayBeShared = true
            }
        }
    },{
        .commandId = CMD_ADD_TWO_NUMBERS
        .params {
            {
                .type = TEE_PARAM_TYPE_VALUE_INPUT
            },{
                .type = TEE_PARAM_TYPE_VALUE_OUTPUT
            }
        }
    }
};

static const TEE_Identity caller = {
    .login = TEE_LOGIN_APPLICATION_USER,
    .uuid =  { 0xfd2a7830U, 0xab65U, 0x565bU,
               { 0xa4U, 0xc8U, 0x80U, 0x32U, 0x43U, 0xf2U, 0xa5U, 0x09U }}
};

TEE_Result TA_CreateEntryPoint(void)
{
    TEE_RegisterCommandPolicy(policy, sizeof(policy) /
                                      sizeof(TEE_InvokeTACommandPolicy));
    TEE_RegisterPermittedCaller(& caller);
    return TEE_SUCCESS;
}
```