Global Platform®

The standard for secure digital services and devices

GlobalPlatform Technology

Annex C: TLS Specification
of TEE Sockets API Specification v1.0.3

Version 1.1.0.13 [target v1.2]

Public Review

January 2024

Document Reference: GPD_SPE_103

THIS SPECIFICATION OR OTHER WORK PRODUCT IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY IMPLEMENTATION OF THIS SPECIFICATION OR OTHER WORK PRODUCT SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE COMPANY, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER DIRECTLY OR INDIRECTLY ARISING FROM THE IMPLEMENTATION OF THIS SPECIFICATION OR OTHER WORK PRODUCT.

Contents

Tables

1 INTRODUCTION

This document includes one annex of TEE Sockets API Specification v1.0.3 ([TEE Sockets]). Additional annexes exist.

The API defined in this specification enables several TLS protocol capabilities. The API supports only client-side TLS functionality.

It is not the role of this specification to guide the reader in determining which TLS protocol capabilities may be safe for their purposes, and this specification recognizes that in some cases the use of weak cryptography by a Trusted Application (TA) may be better than the use of that same cryptography by an application outside of a Trusted Execution Environment (TEE).

GlobalPlatform does provide recommendations for best practices and acceptable cryptography usage. These can be found in GlobalPlatform Cryptographic Algorithm Recommendations ([Crypto Rec]), and relevant sections of that document MAY be applied to the interfaces and API offered by this specification. As always, the developer should refer to appropriate security guidelines.

This annex addresses the instance specification of the Transport Layer Security (TLS) protocol versions 1.3 and 1.2.

GlobalPlatform would like to explicitly encourage readers to contribute to its specifications.

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-GPD_SPE_103@globalplatform.org**

1.1 Audience

This document is suitable for software developers implementing Trusted Applications running inside the Trusted Execution Environment (TEE) which need to make socket networking calls.

This document is also intended for implementers of the TEE itself, its Trusted OS, Trusted Core Framework, the TEE APIs, and the communications infrastructure required to access Trusted Applications.

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	[TEE Core]
GPD_SPE_100	GlobalPlatform Technology TEE Sockets API Specification	[TEE Sockets]
GPD_SPE_101	GlobalPlatform Technology Annex A: TCP/IP Specification of TEE Sockets API Specification	[Sockets TCP/IP]
GPD_SPE_102	GlobalPlatform Technology Annex B: UDP/IP Specification of TEE Sockets API Specification	[Sockets UDP/IP]
GPD_GUI_104	GlobalPlatform Technology Annex D: Examples of Using Interfaces Defined in TEE Sockets API Specification	[Sockets Examples]
GP_TEN_053	GlobalPlatform Technology Cryptographic Algorithm Recommendations	[Crypto Rec]
GP_GUI_001	GlobalPlatform Document Management Guide	[Doc Mgmt]
IANA TLS Cipher Suite Registry	http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml	[IANA]
TLS Cipher Suites	TLS Cipher Suites https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4	[IANA Example]
RFC 2119	Key words for use in RFCs to Indicate Requirement Levels	[RFC 2119]
RFC 4279	PSK Ciphersuites for TLS	[RFC 4279]
RFC 4492	Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)	[RFC 4492]
RFC 5054	Using the Secure Remote Password (SRP) Protocol for TLS Authentication	[RFC 5054]
RFC 5246	The Transport Layer Security (TLS) Protocol	[RFC 5246]
RFC 5280	Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile	[RFC 5280]
RFC 5288	AES Galois Counter Mode (GCM) Cipher Suites for TLS	[RFC 5288]

Standard / Specification	Description	Ref
RFC 5289	TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)	[RFC 5289]
RFC 5487	Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode	[RFC 5487]
RFC 5489	ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)	[RFC 5489]
RFC 5929	Channel Bindings for TLS	[RFC 5929]
RFC 6066	Transport Layer Security (TLS) Extensions: Extension Definition	[RFC 6066]
RFC 6655	AES-CCM Cipher Suites for Transport Layer Security (TLS)	[RFC 6655]
RFC 7301	Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension	[RFC 7301]
RFC 7525	Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)	[RFC 7525]
RFC 7919	Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)	[RFC 7919]
RFC 8174	Amendment to RFC 2119	[RFC 8174]
RFC 8446	The Transport Layer Security (TLS) Protocol Version 1.3	[RFC 8446]
RFC 8447	IANA Registry Updates for TLS and DTLS	[RFC 8447]
RFC 9266	Channel Bindings for TLS 1.3	[RFC 9266]
RFC TBD	Entity Attestation Token Pending publication (draft-ietf-rats-eat-19)	[draft EAT]

1.4 Terminology and Definitions

The following meanings apply to SHALL, SHALL NOT, MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY in this document (refer to [RFC 2119] as amended by [RFC 8174]):

- **SHALL** indicates an absolute requirement, as does **MUST**.
- **SHALL NOT** indicates an absolute prohibition, as does **MUST NOT**.
- **SHOULD** and **SHOULD NOT** indicate recommendations.
- **MAY** indicates an option.

Note that as clarified in the [RFC 8174] amendment, lower case use of these words is not normative.

Selected technical terms used in this document are included in Table 1-2. Additional technical terms are defined in [TEE Sockets] and [TEE Core].

Table 1-2: Terminology and Definitions

Term	Definition
Annex C TEE Sockets TLS API	Short form of, and equivalent to: Annex C: TLS Specification of TEE Sockets API Specification
attestation evidence	See discussion in section C.2.4.
child-most	In a tree, each node except the root is a child of some other node. A “child-most” node has no children of its own. Also known as “leaf” node.
`iSocket`	Interface Socket
`iSocket` instance	Instance of Interface Socket

1.5 Abbreviations

Selected abbreviations and notations used in this document are included in Table 1-3. Additional abbreviations and notations are defined in [TEE Sockets] and [TEE Core].

Table 1-3: Abbreviations

Abbreviation / Notation	Meaning
ALPN	Application-Layer Protocol Negotiation
ASN.1	Abstract Syntax Notation One
DER	Distinguished Encoding Rules
DSS	Digital Signature Standard
ECC	Elliptic Curve Cryptography
GCM	Galois Counter Mode
IP	Internet Protocol
PDC	Pre-Distributed Credentials
PSK	Pre-Shared Key
SPKI	Subject Public Key Info
SRP	Secure Remote Password
TA	Trusted Application
TEE	Trusted Execution Environment
TLS	Transport Layer Security

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 1-4: Revision History

Date	Version	Description
June 2015	1.0	Public Release
Jan 2017	1.0.1	Public Release showing all non-trivial changes since v1.0. • Clarified meaning of one error code
Feb 2021	1.0.2	Public Release showing all non-trivial changes since v1.0. • Clarified limitations on cryptographic recommendations in this specification. Note: Only this annex is being issued as v1.0.2.
Dec 2022	1.1	Changes include: • New functionality and extensions to enable TLS 1.3 client mode • Better operating mode support for TLS key establishment and authentication beyond the original Pre-Shared Keys (PSKs) • Eliminated `TEE_tlsSocket_CertStorageCred` structure and associated unions in `TEE_tlsSocket_Credentials` structure. Note: Only this annex and Annex D ([Sockets Examples]) are being issued as v1.1.
Jun 2023	1.1.0.7	Committee Review
Aug 2023	1.1.0.11	Member Review
Jan 2024	1.1.0.13	Public Review
TBD	1.2	Public Release Changes include: • New functionality and extensions to enable Attestation in the TLS establishment. Note: Only this annex is being issued as v1.2. • TEE Sockets API Specification ([TEE Sockets]) remains at v1.0.3. • Annex A ([Sockets TCP/IP]) and Annex B ([Sockets UDP/IP]) remain at v1.0.1. • Annex D ([Sockets Examples]) remains at v1.1.

Annex C TEE_tlsSocket INSTANCE SPECIFICATION

This annex specifies the TEE_iSocket interface for the Transport Layer Security (TLS) protocol. Implementation of TLS protocol support within the TEE is optional. If the TLS protocol is implemented, the implementation SHALL reside wholly within the TEE because it alters the security level of the information passing over the socket.

C.1 General Information

C.1.1 Header File Name

The corresponding header file SHALL be named **"tee_tlssocket.h"**.

C.1.1.1 API Version

Since: Annex C TEE Sockets TLS API v1.1

The header file SHALL contain version specific definitions from which TA compilation options can be selected.

```
#define TEE_SOCKET_TLS_API_MAJOR_VERSION ([Major version number])
#define TEE_SOCKET_TLS_API_MINOR_VERSION ([Minor version number])
#define TEE_SOCKET_TLS_API_MAINTENANCE_VERSION ([Maintenance version number])
#define TEE_SOCKET_TLS_API_VERSION (TEE_SOCKET_TLS_API_MAJOR_VERSION << 24) +
(TEE_SOCKET_TLS_API_MINOR_VERSION << 16) +
(TEE_SOCKET_TLS_API_MAINTENANCE_VERSION << 8)
```

The document version-numbering format is ***X.Y[.z]***, where:

Major Version (X) is a positive integer identifying the major release.

Minor Version (Y) is a positive integer identifying the minor release.

The optional Maintenance Version (z) is a positive integer identifying the maintenance release.

TEE_SOCKET_TLS_API_MAJOR_VERSION indicates the major version number of the TEE Sockets TLS API. It SHALL be set to the major version number of this specification.

TEE_SOCKET_TLS_API_MINOR_VERSION indicates the minor version number of the TEE Sockets TLS API. It SHALL be set to the minor version number of this specification. If the minor version is zero, then one zero SHALL be present.

TEE_SOCKET_TLS_API_MAINTENANCE_VERSION indicates the maintenance version number of the TEE Sockets TLS API. It SHALL be set to the maintenance version number of this specification. If the maintenance version is zero, then one zero SHALL be present.

The definitions of "Major Version", "Minor Version", and "Maintenance Version" in the version number of this specification are determined as defined in the GlobalPlatform Document Management Guide ([Doc Mgmt]). In particular, the value of TEE_SOCKET_TLS_API_MAINTENANCE_VERSION SHALL be zero if it is not already defined as part of the version number of this document. The "Draft Revision" number SHALL NOT be provided as an API version indication.

A compound value SHALL also be defined. If the Maintenance version number is `0`, the compound value SHALL be defined as:

```
#define TEE_SOCKET_TLS_API_[Major version number]_[Minor version number]
```

If the Maintenance version number is not zero, the compound value SHALL be defined as:

```
#define TEE_SOCKET_TLS_API_[Major version number]_[Minor version
number]_[Maintenance version number]
```

Some examples of version definitions:

For GlobalPlatform TEE Sockets TLS API Specification v1.3, these would be:

```
#define TEE_SOCKET_TLS_API_MAJOR_VERSION        (1)
#define TEE_SOCKET_TLS_API_MINOR_VERSION        (3)
#define TEE_SOCKET_TLS_API_MAINTENANCE_VERSION  (0)
#define TEE_SOCKET_TLS_API_1_3
```

And the value of `TEE_SOCKET_TLS_API_VERSION` would be `0x01030000`.

For a maintenance release of the specification as v2.14.7, these would be:

```
#define TEE_SOCKET_TLS_API_MAJOR_VERSION        (2)
#define TEE_SOCKET_TLS_API_MINOR_VERSION        (14)
#define TEE_SOCKET_TLS_API_MAINTENANCE_VERSION  (7)
#define TEE_SOCKET_TLS_API_2_14_7
```

And the value of `TEE_SOCKET_TLS_API_VERSION` would be `0x020E0700`.

C.1.2 Specification Version Number Property

This specification defines a TEE property containing the version number of the specification the implementation conforms to. The property can be retrieved using the normal Property Access Functions defined in TEE Internal Core API Specification ([TEE Core]). The property SHALL be named **"`gpd.tee.sockets.tls.version`"** and SHALL be of `integer` type with the interpretation given in TEE Sockets API Specification ([TEE Sockets]) section 4.2.

The `iSocket` interface variable `TEE_iSocketVersion` indicates which version of the `iSocket` interface (defined in [TEE Sockets] section 5) this protocol's `iSocket struct` conforms to.

C.1.3 Protocol Identifier Value

The assigned protocol identifier for `TEE_ISOCKET_PROTOCOLID_TLS` is **103** (decimal) or **0x67** (hex).

C.1.4 Panic Numbering

The Specification Number for reporting Panics from the TLS instance of the `iSocket` API SHALL be **103**.

The Function Numbers for reporting Panics are defined in [TEE Sockets] section 4.4.

C.2 Transport Layer Security (TLS)

TLS is a client-server secure channel protocol that can be layered on top of a connection-oriented, reliable transport protocol, such as TCP. Therefore, a TLS socket MAY be layered on top of a TCP socket (defined in Annex A [Sockets TCP/IP]), but SHALL NOT be layered on top of a UDP socket (defined in Annex B [Sockets UDP/IP]). The API defined in this specification SHALL be used to establish client-side TLS endpoints only.

TLS consists of two main components: the *handshake protocol*, which provides authenticated key exchange and the *record protocol* which provides confidentiality, integrity, and replay protection.

C.2.1 Handshake Variants

The implementation SHALL support server-authenticated TLS handshake, where the client SHALL authenticate the server using a public key certificate and a proof-of-possession of the corresponding private key.

Additionally, the implementation MAY support the following types of TLS handshake:

- Mutually authenticated handshake – In this handshake type, the client SHALL authenticate the server as above, and in addition the client SHALL authenticate itself to the server via a public key certificate and proof-of-possession of the corresponding private key.
- PSK-authenticated handshake – In this handshake type, the endpoints SHALL be authenticated via proof-of-possession of an externally provisioned Pre-Shared Key (PSK).
- Resumed handshake – In this handshake type, the client SHALL present to the server an encrypted session ticket containing the state of a previous TLS session. The previous session is then resumed and expensive public key cryptography (authentication and key exchange) can be skipped.

A Trusted Application (TA) SHALL use the `gpd.tee.tls.handshake` property to identify the available handshake types. The value of `gpd.tee.tls.handshake` is a `uint32_t` indicating the TLS handshake types that the underlying TEE supports. Table C-1 defines the bit-mask constants for `gpd.tee.tls.handshake`.

Table C-1: `gpd.tee.tls.handshake` Property Bit-mask Constants

Name	Value
`TEE_TLS_HANDSHAKE_TYPE_SERVER_AUTHENTICATE_ONLY`	`0x00000000`
`TEE_TLS_HANDSHAKE_TYPE_MUTUAL_AUTHENTICATED`	`0x00000001`
`TEE_TLS_HANDSHAKE_TYPE_PSK_AUTHENTICATED`	`0x00000002`
`TEE_TLS_HANDSHAKE_TYPE_RESUMED`	`0x00000004`
Reserved for GlobalPlatform use	`0x007FFFF8`
`TEE_TLS_HANDSHAKE_TYPE_ILLEGAL_VALUE`	`0x00800000`
Implementation defined	`0xFF000000`

`TEE_TLS_HANDSHAKE_TYPE_ILLEGAL_VALUE` is reserved for testing and validation and SHALL be treated as an undefined value when the corresponding bit is set in the value retrieved as the `gpd.tee.tls.handshake` property.

Note: `TEE_TLS_HANDSHAKE_TYPE_SERVER_AUTHENTICATE_ONLY` indicates that the underlying TLS implementation does not support any of the additional handshake type. In this case, the TA SHALL only use server-authenticated TLS handshake. Regardless of the `gpd.tee.tls.handshake` property value, the implementation SHALL always support server-authenticated TLS handshake.

C.2.2 Credentials and Authentication

C.2.2.1 Server (Remote Endpoint) Authentication

This specification SHALL support at least one of the following credentials for server (remote endpoint) authentication:

- X.509 certificates – In this variant, the TA SHALL provide one or more trusted certificates as Pre-Distributed Credentials (PDCs). The implementation SHALL validate the server's certificate chain received during the TLS handshake against the PDCs provided by the TA. If the chain contains the trusted certificate (either as the root certificate, intermediate certificate, or child-most certificate), validation SHALL be deemed successful.
- Certificate and public key pinning – When using pinning, the TA SHALL provide as PDC at least one trusted SHA-256 hash of server end-entity certificates or the `SubjectPublicKeyInfo` (SPKI) structures of the certificates. The TA MAY also provide as PDC a list of trusted SHA-256 hashes of server end-entity certificates or the SPKI structures of the certificates. The implementation SHALL consider peer authentication successful if the hash of the received certificate or SPKI matches one of the pinned values and the peer's `CertificateVerify` signature can be validated successfully using the corresponding public key.
- PSKs – When using PSK authentication, the TA SHALL provide as PDCs a PSK value and a PSK identity used to identify the PSK to be used in the TLS connection. Note that in order to use a PSK in TLS 1.2, the TA SHALL have enabled at least one cipher suite whose name starts with `TEE_TLS_PSK`. In TLS 1.3, there is no such restriction, as PSKs can be used with all TLS 1.3 cipher suites. If the PSK was derived in an earlier TLS 1.3 handshake, the client MAY later provide the corresponding server-encrypted session ticket to resume the earlier session. If the PSK is used for TLS 1.3 session resumption, PSK identity MAY NOT be provided.
- Secure Remote Password (SRP) ([RFC 5054]) – SRP SHALL only be used for TLS 1.2. Note that in order to use SRP, the TA SHALL enable at least one cipher suite whose name starts with `TEE_TLS_SRP`.
- Legacy pre-distributed server public key authentication – In this variant, the TA SHALL provide as PDC the public key of the server and SHALL use it for all encryptions and verifications of server messages. The public key in the certificate sent by the server during the handshake is ignored. This option is provided for interoperability purposes and SHALL only be used for TLS 1.2 implementations.

TA SHALL use the `gpd.tee.tls.auth.remote.credential` property to identify the available credential types for authenticating remote endpoints. The value of `gpd.tee.tls.auth.remote.credential` is a `uint32_t` indicating the authentication types that the underlying TEE supports for remote endpoint authentication. Table C-2 defines the bit-mask constants for remote credential types.

203 **Table C-2: `gpd.tee.tls.auth.remote.credential` Property Bit-mask Constants**

Name	Value
`TEE_TLS_AUTH_REMOTE_CREDENTIAL_NONE`	`0x00000000`
`TEE_TLS_AUTH_REMOTE_CREDENTIAL_PDC`	`0x00000001`
`TEE_TLS_AUTH_REMOTE_CREDENTIAL_X509_CERT`	`0x00000002`
`TEE_TLS_AUTH_REMOTE_CREDENTIAL_CERT_PINNING`	`0x00000004`
`TEE_TLS_AUTH_REMOTE_CREDENTIAL_PSK`	`0x00000008`
`TEE_TLS_AUTH_REMOTE_CREDENTIAL_SRP`	`0x00000010`
Reserved for GlobalPlatform use	`0x007FFFE0`
`TEE_TLS_AUTH_REMOTE_CREDENTIAL_ILLEGAL_VALUE`	`0x00800000`
Implementation defined	`0xFF000000`

204

205 `TEE_TLS_AUTH_REMOTE_CREDENTIAL_ILLEGAL_VALUE` is reserved for testing and validation and SHALL
206 be treated as an undefined value when the corresponding bit is set in the value retrieved as the
207 `gpd.tee.tls.auth.remote.credential` property.

208 *Note:* `TEE_TLS_AUTH_REMOTE_CREDENTIAL_NONE` SHALL be treated as an error.

209

210 C.2.2.2 Client (Local Endpoint) Authentication

211 Client authentication is optional, but if client authentication is supported, then the implementation SHALL
212 support the following client authentication method:

213 • Private key and X.509 certificate – In this variant, the TA SHALL provide as PDCs a handle to a
214 private key in trusted storage, plus a certificate chain where the child-most certificate contains the
215 public key counterpart. The chain may consist of one or more certificates. The implementation sends
216 the certificate to the server during the handshake for validation. Note that when using TLS 1.2, the TA
217 SHALL enable at least one cipher suite that matches the type of the provided private key. For
218 example, to use an ECDSA keypair for authentication in TLS 1.2, the caller could enable any of the
219 cipher suites whose name starts with `TEE_TLS_ECDHE_ECDSA`. In TLS 1.3, there are no such
220 restrictions, and all supported key types MAY be used with any TLS 1.3 cipher suite.

221 Additionally, the implementation MAY support the following client authentication methods:

222 • PSKs (See remarks in section C.2.2.1.)

223 • Secure Remote Password (SRP) ([RFC 5054]) – This variant can be used for TLS 1.2 only.

TA SHALL use the `gpd.tee.tls.auth.local.credential` property to identify the available credential types for client authentication. The value of `gpd.tee.tls.auth.local.credential` is a `uint32_t` indicating the authentication types that the underlying TEE supports for client authentication. Table C-3 defines the bit-mask constants for local credential types.

Table C-3: `gpd.tee.tls.auth.local.credential` Property Bit-mask Constants

Name	Value
`TEE_TLS_AUTH_LOCAL_CREDENTIAL_NONE`	`0x00000000`
`TEE_TLS_AUTH_LOCAL_CREDENTIAL_X509`	`0x00000001`
`TEE_TLS_AUTH_LOCAL_CREDENTIAL_PSK`	`0x00000002`
`TEE_TLS_AUTH_LOCAL_CREDENTIAL_SRP`	`0x00000004`
Reserved for GlobalPlatform use	`0x007FFFF8`
`TEE_TLS_AUTH_LOCAL_CREDENTIAL_ILLEGAL_VALUE`	`0x00800000`
Implementation defined	`0xFF000000`

`TEE_TLS_AUTH_LOCAL_CREDENTIAL_ILLEGAL_VALUE` is reserved for testing and validation and SHALL be treated as an undefined value when the corresponding bit is set in the value retrieved as the `gpd.tee.tls.auth.local.credential` property.

Note: `TEE_TLS_AUTH_LOCAL_CREDENTIAL_NONE` indicates that the underlying TLS implementation does not support client authentication.

For session resumption, the TA SHALL provide a storage area for the encrypted session ticket it receives from the server at the end of a standard handshake.

C.2.3 TLS Extensions and Optional Features

Section 4.2 in [RFC 8446] and section 7.4.1.4 in [RFC 5246] define a set of TLS protocol extensions and associated extension messages. Some extensions are mandatory in certain TLS protocol versions. For example, `supported_versions` is mandatory when TLS 1.3 is offered in the handshake. Other extensions are mandatory in certain handshake variants. For example, `key_share` is mandatory in TLS 1.3 handshakes that use (EC)DH key exchange. Also, optional protocol features exist that are not associated with an extension. One such example is client authentication. This section provides an overview of extensions and optional protocol features supported in this specification.

The table below provides an overview of extensions and options relevant to this specification. The implementation SHALL support the extensions and optional features marked as “mandatory” in the table. The implementation MAY support further extensions and features if needed.

Table C-4: TLS Extensions and Options Relevant to this Specification

Extension/Optional Feature	TLS 1.3	TLS 1.2	Notes
`server_name`	Mandatory	Mandatory	TA can influence the extension contents. (See section C.2.7.)

Extension/Optional Feature	TLS 1.3	TLS 1.2	Notes
`supported_versions`	Mandatory	Optional, but recommended	TA can influence the extension contents. (See section C.2.6.1.) [RFC 8446] recommends that the extension is sent even when only TLS 1.2 and below is supported. For a dual-stack TLS client implementation, a `ClientHello` message would contain the `supported_version` extension and a TLS 1.2-only server implementation would lead to a fallback to TLS 1.2 even if the server does not understand the `supported_version` extension (or any other TLS 1.3 extensions).
`supported_groups`	Mandatory for (EC)DH handshakes	Optional, but can be used to indicate ECC curves only	TA can influence the extension contents. (See section C.2.6.5.)
`signature_algorithms`	Mandatory for certificate-authenticated handshakes	Optional, but recommended	TA can influence the extension contents. (See section C.2.6.4.)
`signature_algorithms_cert`	Optional	Not defined	TA can influence the extension contents. (See sections C.2.6.4 and C.2.7.)
`key_share`	Mandatory for (EC)DH handshakes	Not defined	
`pre_shared_key`	Mandatory for PSK handshakes and resumed handshakes	Not defined	
`max_fragment_length`	Optional	Optional	The implementation MAY send this extension according to requirements such as memory constraints. This specification does not provide an API that would allow the TA to influence the extension.

Extension/Optional Feature	TLS 1.3	TLS 1.2	Notes
`application_layer_protocol_negotiation`	Optional	Optional	TA can influence the extension contents (see section C.2.7).
Client authentication	Optional	Optional	See section C.2.6.9.
Post-handshake client authentication	Optional	Not defined	The implementation MAY support post-handshake client authentication if the TA has provided a private key and a certificate in the client PDC structure. (See section C.2.6.9.)
Renegotiation	Not defined	Optional, but not recommended	If renegotiation is supported by the implementation, then the necessary countermeasures to known attacks SHALL also be supported. Such countermeasures include those listed in [RFC 7525] section 3.5. For example, the `renegotiation_info` extension SHALL be sent when the implementation supports renegotiation.
Ticket-based session resumption	Optional	Optional	See section C.2.6.7.
PSK handshakes with externally established PSK	Optional	Optional	
0-RTT early data	SHOULD NOT be used	Not defined	0-RTT data is not forward-secret or replay-protected by default. Replayable 0-RTT data presents a number of security threats to TLS-using applications, unless those applications are specifically engineered to be safe under replay. This specification provides no API for the TA to supply early data to the implementation.
Record padding	Optional	Not defined	This specification does not provide an API that would allow the TA to influence the use of record padding.
Remote attestation	Optional	Optional	See section C.2.4.

250

C.2.4 Remote Attestation

Remote attestation allows endpoints to prove their security properties to relying parties by generating and transmitting attestation evidence. The evidence contains claims that are approved and signed by an attesting environment, which the relying party assumes to be more trustworthy than the target of the attestation. Examples of attesting environments include the TEE or a special TA providing attestation services. The attestation claims could contain, for example, an identity of the TA and the TEE, and information about the security features and security level provided by the TEE.

This specification supports both intra-handshake and post-handshake attestation. In the intra-handshake variant, attestation occurs within the TLS handshake, and the TLS session is established only if attestation is successful. In the post-handshake variant, attestation is performed over an already established TLS session using a separate protocol.

In all variants, attestation evidence is required to contain channel bindings that are unique to the TLS handshake, providing strong protection against relay attacks.[1]

Note on terminology: In some specifications, such as the Entity Attestation Token ([draft EAT]), attestation evidence (a signed message containing attestation claims) is called simply “an attestation”. The current document follows the terminology used in the specifications of the IETF’s Remote Attestation Architectures (RATS) Working Group, such as RFC 9334.

C.2.4.1 Post-handshake Attestation

This specification defines an API (see section C.2.9) that can be used to retrieve a value that is unique to an established TLS session. This value can then be used as the channel bindings in a subsequent remote attestation protocol. When generating evidence, the value can be included as one of the signed claims. When verifying evidence, the value can be used as a reference against which the channel bindings extracted from the received evidence can be compared.

C.2.4.2 Intra-handshake Attestation

The drawbacks of post-handshake attestation include the requirement for a custom remote attestation protocol and an additional round-trip to request and transmit attestation evidence. No standard remote attestation protocol exists that TAs could readily use. For these reasons, it is often preferable to transmit attestation evidence within the TLS handshake.

This specification provides an API (see sections C.2.6.16 through C.2.6.18) that allows binding remote attestation to TLS session establishment. Currently, three methods for transmitting attestation evidence in a TLS handshake are supported:

- X.509 extension: Evidence is transmitted in an X.509 extension in the TLS endpoint authentication certificate.

[1] In a relay attack, an attacker has access to a compromised device A and an uncompromised device B. When A receives an attestation request over communication channel X, the attacker opens a second channel (Y) to B and forwards the request to B. The device B then sends a valid attestation evidence over channel Y to the attacker, who relays the evidence to channel X, thus successfully attesting the compromised device A. To prevent such attacks, evidence should contain a unique communication channel identifier (channel bindings) so that the receiver can verify that the evidence was meant to be transmitted over the current channel.

- Extra certificate: Evidence is transmitted in an extra certificate appended to the TLS endpoint authentication certificate chain.
- Certificate message extension: Evidence is transmitted in a TLS extension in the attester's Certificate handshake message.

All of these approaches use standard TLS extension mechanisms and are fully compatible with the TLS specification.

The API defined in this specification allows both sending and verifying attestation evidence. The API allows specifying a trust anchor that the implementation shall use to verify the evidence signature. The implementation is also required to validate the channel bindings in the evidence. If the verification of the evidence signature or the channel bindings fails, the implementation is required to terminate the TLS handshake.

Since it is conceivable that the implementation cannot appraise all attestation claims in the received evidence, this specification provides an API (see section C.2.9) that the TA can use to retrieve the evidence for further self-appraisal.

C.2.4.3 Scope of the Attestation Feature

This specification focuses on providing an API that TAs can use to send, receive, and verify attestation evidence such that the evidence is cryptographically bound to a TLS session. Since there is currently no standard for the use of attestation in TLS, a major goal of this specification is to allow writing TAs that can make use of implementation-defined attestation extensions and formats, such that the TAs need not be rewritten when the implementation later switches to a standard variant of attested TLS.

This specification does not specify the format or contents of the attestation evidence, except that the evidence is required to contain channel bindings. Possibilities include using the Entity Attestation Token ([draft EAT]) as the evidence format, or using an ASN.1 type such as the `TCBInfo` defined by the Trusted Computing Group. The implementation does, however, allow the TA to extract the received attestation evidence so that any format or use case specific validation steps can be performed.

This specification does not define the format of the attestation request that the implementation may send. Standardization activities are ongoing in this field, for example in the IETF (draft-fossati-tls-attestation[2]). However, no stable and widely supported specification is available at this time.

The implementation is free to use any evidence format and attestation request extension. For interoperability, it is assumed that the TA will only attempt remote attestation with a remote endpoint that it knows to support an evidence format the implementation is able to provide and to verify. The implementation SHOULD provide an implementation defined way such as a TEE property that the TA can use to find out which evidence formats and TLS extensions for attestation the implementation supports.

[2] https://datatracker.ietf.org/doc/html/draft-fossati-tls-attestation-03

C.2.4.4 Channel Bindings

This specification requires the use of channel bindings in both transmitted and received attestation evidence. This subsection specifies how the channel bindings should be computed.

The TLS 1.3 specification ([RFC 8446] section 7.5) defines TLS-Exporter mechanism, which can be used to derive handshake-dependent secret values. These values can be used as channel bindings for TLS 1.3 and 1.2, as specified in [RFC 9266].

In TLS 1.3 the value can be based on either the exporter master secret or the early exporter master secret. [RFC 8446] requires that the early exporter master secret be used only when 0-RTT data is transmitted. Since this specification does not support 0-RTT data, the early exporter master secret SHALL NOT be used to derive channel bindings. Instead, if the exporter master secret has not yet been computed[3] at the time when attestation evidence is to be generated, a transcript hash shall be used instead. The hash SHALL cover the handshake through the message where attestation evidence is transmitted. The hash algorithm shall be the same as the hash algorithm defined by the negotiated cipher suite.

In summary, channel bindings for attestation evidence SHALL be computed as follows:

Table C-5: Computing Channel Bindings

Protocol Version	Endpoint that Generates Evidence	Channel Bindings
TLS 1.3	Client	"`tls-exporter`" ([RFC 9266])
TLS 1.3	Server	Transcript hash over `ClientHello` through the later of `EncryptedExtensions` or `CertificateRequest`
TLS 1.2	Client	"`tls-exporter`" ([RFC 9266]). Note that the extended master secret extension SHALL be used in the handshake.
TLS 1.2	Server	Transcript hash over `ClientHello` through `ServerHello`

C.2.5 TEE_iSocket Instance Variable for TLS

```
extern TEE_iSocket * const TEE_tlsSocket;
```

The name of the instance variable for the TLS sockets interface SHALL be `TEE_tlsSocket`.

[3] In all evidence transmission methods supported by this specification, evidence is transmitted inside the Certificate handshake messages and the server's Certificate message is sent before the exporter master secret is available.

C.2.6 Type Definitions

The header file SHALL provide the following constants and structures.

The implementation SHALL support the subset of TLS 1.3 or TLS 1.2 defined in this document. The implementation MAY support both TLS 1.3 and TLS 1.2.

A compliant implementation MAY support further TLS options and algorithms; as this is implementation specific, it will provide an implementation specific methodology to indicate this extension.

A particular TLS socket may be configured by the TA to restrict itself by supplying a specific version (e.g. `TEE_TLS_VERSION_1v2`, `TEE_TLS_VERSION_1v3`), or a combination (e.g. `TEE_TLS_VERSION_1v2 | TEE_TLS_VERSION_1v3`). An implementation may also indicate that it supports all TLS versions (`TEE_TLS_VERSION_ALL`); however, the use of `TEE_TLS_VERSION_ALL` is not recommended.

C.2.6.1 TEE_tlsSocket_TlsVersion

Since: Annex C TEE Sockets TLS API v1.1 – See Backward Compatibility note below.

```
typedef uint32_t TEE_tlsSocket_TlsVersion;
```

The `TEE_tlsSocket_TlsVersion` type is a bit-mask indicating the TLS versions the endpoint supports. Table C-6 defines the values of `TEE_tlsSocket_TlsVersion`.

If multiple versions are enabled and the highest version is TLS 1.2, then the implementation SHALL advertise the highest enabled version in the `client_version` field of the `ClientHello` message. If TLS 1.3 is enabled, the implementation SHALL send the enabled versions, from highest to lowest order, in the `supported_versions` extension of the `ClientHello` message.

Table C-6: `TEE_tlsSocket_TlsVersion` Bit-mask Constants

Name	Value	Meaning
`TEE_TLS_VERSION_ALL`	`0x00000000`	Accept connections to servers using any TLS version supported by the implementation
`TEE_TLS_VERSION_1v2`	`0x00000001`	Accept connections to servers using TLS 1.2
`TEE_TLS_VERSION_PRE1v2`	`0x00000002`	Accept connections to server using a TLS version prior to TLS 1.2
`TEE_TLS_VERSION_1v3`	`0x00000004`	Accept connections to servers using TLS 1.3
Reserved for GlobalPlatform use	`0x007FFFF8`	Set bits reserved for use by GlobalPlatform
`TEE_TLS_VERSION_ILLEGAL_VALUE`	`0x00800000`	Reserved for testing and validation and SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_Setup` structure or the `TEE_tlsSocket_SessionInfo` structure.
Implementation defined	`0xFF000000`	Set bits reserved for implementation defined flags. Used to assign specific handshakes or methods.

Backward Compatibility

Prior to Annex C TEE Sockets TLS API v1.1, `TEE_tlsSocket_TlsVersion` was defined as an `enum`.

C.2.6.2 TEE_tlsSocket_CipherSuites_GroupA

Since: Annex C TEE Sockets TLS API v1.1 – See Backward Compatibility note below.

```
typedef uint32_t *TEE_tlsSocket_CipherSuites_GroupA;
```

The `TEE_tlsSocket_CipherSuites_GroupA` type defines the IANA TLS Cipher Suite constants ([IANA]) that are supported for TLS 1.2. Table C-7 defines the values of `TEE_tlsSocket_CipherSuites_GroupA`.

In TLS 1.2, the cipher suite defines the used key exchange, authentication, symmetric encryption, and hash algorithms, using the following cipher suite naming scheme:

```
TEE_TLS_[keyex alg]_[auth alg]_[symmetric alg]_[hash]
```

It is the responsibility of the TA to choose cipher suites that are compatible with the rest of the configuration.

Table C-7: `TEE_tlsSocket_CipherSuites_GroupA` Values

Algorithm	Value	Main Reference
TEE_TLS_NULL_WITH_NULL_NULL	0x00000000	List Termination
TEE_TLS_RSA_WITH_3DES_EDE_CBC_SHA	0x0000000A	[RFC 5246]
TEE_TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	0x00000013	
TEE_TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	0x00000016	
TEE_TLS_RSA_WITH_AES_128_CBC_SHA	0x0000002F	
TEE_TLS_DHE_DSS_WITH_AES_128_CBC_SHA	0x00000032	
TEE_TLS_DHE_RSA_WITH_AES_128_CBC_SHA	0x00000033	
TEE_TLS_RSA_WITH_AES_256_CBC_SHA	0x00000035	
TEE_TLS_DHE_DSS_WITH_AES_256_CBC_SHA	0x00000038	
TEE_TLS_DHE_RSA_WITH_AES_256_CBC_SHA	0x00000039	
TEE_TLS_RSA_WITH_AES_128_CBC_SHA256	0x0000003C	
TEE_TLS_RSA_WITH_AES_256_CBC_SHA256	0x0000003D	
TEE_TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	0x00000040	
TEE_TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	0x00000067	
TEE_TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	0x0000006A	
TEE_TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	0x0000006B	
TEE_TLS_PSK_WITH_3DES_EDE_CBC_SHA	0x0000008B	[RFC 4279]
TEE_TLS_PSK_WITH_AES_128_CBC_SHA	0x0000008C	
TEE_TLS_PSK_WITH_AES_256_CBC_SHA	0x0000008D	
TEE_TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA	0x0000008F	
TEE_TLS_DHE_PSK_WITH_AES_128_CBC_SHA	0x00000090	
TEE_TLS_DHE_PSK_WITH_AES_256_CBC_SHA	0x00000091	
TEE_TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA	0x00000093	

Algorithm	Value	Main Reference
TEE_TLS_RSA_PSK_WITH_AES_128_CBC_SHA	0x00000094	
TEE_TLS_RSA_PSK_WITH_AES_256_CBC_SHA	0x00000095	
TEE_TLS_RSA_WITH_AES_128_GCM_SHA256	0x0000009C	[RFC 5288]
TEE_TLS_RSA_WITH_AES_256_GCM_SHA384	0x0000009D	
TEE_TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	0x0000009E	
TEE_TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	0x0000009F	
TEE_TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	0x000000A2	
TEE_TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	0x000000A3	
TEE_TLS_PSK_WITH_AES_128_GCM_SHA256	0x000000A8	[RFC 5487]
TEE_TLS_PSK_WITH_AES_256_GCM_SHA384	0x000000A9	
TEE_TLS_DHE_PSK_WITH_AES_128_GCM_SHA256	0x000000AA	
TEE_TLS_DHE_PSK_WITH_AES_256_GCM_SHA384	0x000000AB	
TEE_TLS_RSA_PSK_WITH_AES_128_GCM_SHA256	0x000000AC	
TEE_TLS_RSA_PSK_WITH_AES_256_GCM_SHA384	0x000000AD	
TEE_TLS_PSK_WITH_AES_128_CBC_SHA256	0x000000AE	
TEE_TLS_PSK_WITH_AES_256_CBC_SHA384	0x000000AF	
TEE_TLS_DHE_PSK_WITH_AES_128_CBC_SHA256	0x000000B2	
TEE_TLS_DHE_PSK_WITH_AES_256_CBC_SHA384	0x000000B3	
TEE_TLS_RSA_PSK_WITH_AES_128_CBC_SHA256	0x000000B6	
TEE_TLS_RSA_PSK_WITH_AES_256_CBC_SHA384	0x000000B7	
TEE_TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	0x0000C008	[RFC 4492]
TEE_TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	0x0000C009	
TEE_TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	0x0000C00A	
TEE_TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	0x0000C012	
TEE_TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	0x0000C013	
TEE_TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	0x0000C014	
TEE_TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA	0x0000C01A	[RFC 5054]
TEE_TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA	0x0000C01B	
TEE_TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA	0x0000C01C	
TEE_TLS_SRP_SHA_WITH_AES_128_CBC_SHA	0x0000C01D	
TEE_TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA	0x0000C01E	
TEE_TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA	0x0000C01F	
TEE_TLS_SRP_SHA_WITH_AES_256_CBC_SHA	0x0000C020	
TEE_TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA	0x0000C021	

Algorithm	Value	Main Reference
TEE_TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA	0x0000C022	
TEE_TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	0x0000C023	[RFC 5289]
TEE_TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	0x0000C024	
TEE_TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	0x0000C027	
TEE_TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	0x0000C028	
TEE_TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	0x0000C02B	
TEE_TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	0x0000C02C	
TEE_TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	0x0000C02F	
TEE_TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	0x0000C030	
TEE_TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA	0x0000C034	
TEE_TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA	0x0000C035	
TEE_TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA	0x0000C036	
TEE_TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256	0x0000C037	
TEE_TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384	0x0000C038	
TEE_TLS_RSA_WITH_AES_128_CCM	0x0000C09C	[RFC 6655]
TEE_TLS_RSA_WITH_AES_256_CCM	0x0000C09D	
TEE_TLS_DHE_RSA_WITH_AES_128_CCM	0x0000C09E	
TEE_TLS_DHE_RSA_WITH_AES_256_CCM	0x0000C09F	
TEE_TLS_PSK_WITH_AES_128_CCM	0x0000C0A4	
TEE_TLS_PSK_WITH_AES_256_CCM	0x0000C0A5	
TEE_TLS_DHE_PSK_WITH_AES_128_CCM	0x0000C0A6	
TEE_TLS_DHE_PSK_WITH_AES_256_CCM	0x0000C0A7	
Private use	0x0000FF00-0x0000FFFF	[RFC 8447]
TEE_TLS_CIPHERSUITES_GROUPA_ILLEGAL_VALUE	0x00007FFF	

`TEE_TLS_CIPHERSUITES_GROUPA_ILLEGAL_VALUE` is reserved for testing and validation and SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_Setup` structure.

All values not listed in the table are reserved for future use.

Backward Compatibility

Prior to Annex C TEE Sockets TLS API v1.1, `TEE_tlsSocket_CipherSuites` was defined as an `enum`.

C.2.6.3 TEE_tlsSocket_CipherSuites_GroupB

Since: Annex C TEE Sockets TLS API v1.1

```
typedef uint32_t * TEE_tlsSocket_CipherSuites_GroupB;
```

The `TEE_tlsSocket_CipherSuites_GroupB` type defines the IANA TLS Cipher Suite constants ([IANA]) that are supported for TLS 1.3. Table C-8 defines the values of `TEE_tlsSocket_CipherSuites_GroupB`.

In TLS 1.3, the cipher suite defines the used symmetric algorithm and handshake hash algorithm. Key exchange and authentication algorithms must be chosen separately; see sections C.2.6.4 and C.2.6.5.

Table C-8: `TEE_tlsSocket_CipherSuites_GroupB` Values

Algorithm	Value	Main Reference
`TEE_TLS_NULL_WITH_NULL_NULL`	`0x00000000`	List Termination
Reserved for GlobalPlatform use	`0x00000001 – 0x00001300`	
`TEE_TLS_AES_128_GCM_SHA256`	`0x00001301`	[RFC 8446]
`TEE_TLS_AES_256_GCM_SHA384`	`0x00001302`	
`TEE_TLS_CHACHA20_POLY1305_SHA256` (see below)	`0x00001303`	
`TEE_TLS_AES_128_CCM_SHA256`	`0x00001304`	
`TEE_TLS_AES_128_CCM_8_SHA256`	`0x00001305`	
`TEE_TLS_CIPHERSUITES_GROUPB_ILLEGAL_VALUE`	`0x00007FFF`	
Reserved for private use	`0x0000FF00 - 0x0000FFFF`	[RFC 8447]

`TEE_TLS_CIPHERSUITES_GROUPB_ILLEGAL_VALUE` is reserved for testing and validation and SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_Setup` structure.

All values not listed in the table are reserved for future use. However, an implementation MAY extend this table according to the values defined by IANA; see e.g. [IANA Example].

`TEE_TLS_CHACHA20_POLY1305_SHA256` is optional unless Poly1305 and ChaCha20 are mandated in [TEE Core].

C.2.6.4 TEE_tlsSocket_SignatureScheme

Since: Annex C TEE Sockets TLS API v1.1

```
typedef uint32_t TEE_tlsSocket_SignatureScheme;
```

The `TEE_tlsSocket_SignatureScheme` type defines the IANA TLS Signature Scheme ([IANA]) constants that are supported. Table C-9 defines the values of `TEE_tlsSocket_SignatureScheme`.

The array SHALL include only signature algorithms supported by the TEE (see [TEE Core] Table 6-11). To determine whether the TEE supports a particular signature algorithm, the TA can use the `TEE_IsAlgorithmSupported` function (see [TEE Core] section 6.2.9). If the list contains an algorithm the implementation does not support, the implementation SHALL return the `TLS_ISOCKET_TLS_ERROR_UNSUPPORTED_SIGALG` error code.

The provided list SHALL be sent by the implementation to the server in the `signature_algorithms` extension of the `ClientHello` message.

Table C-9: `TEE_tlsSocket_SignatureScheme` Values

Algorithm Group	Algorithm	Value
RSASSA-PKCS1-v1_5	TEE_TLS_RSA_PKCS1_SHA256	0x00000401
	TEE_TLS_RSA_PKCS1_SHA384	0x00000501
	TEE_TLS_RSA_PKCS1_SHA512	0x00000601
ECDSA	TEE_TLS_ECDSA_SECP256R1_SHA256	0x00000403
	TEE_TLS_ECDSA_SECP384R1_SHA384	0x00000503
	TEE_TLS_ECDSA_SECP521R1_SHA512	0x00000603
RSASSA-PSS with public key OID rsaEncryption	TEE_TLS_RSA_PSS_RSAE_SHA256	0x00000804
	TEE_TLS_RSA_PSS_RSAE_SHA384	0x00000805
	TEE_TLS_RSA_PSS_RSAE_SHA512	0x00000806
EdDSA	TEE_TLS_ED25519	0x00000807
	TEE_TLS_ED448	0x00000808
RSASSA-PSS with public key OID RSASSA-PSS	TEE_TLS_RSA_PSS_PSS_SHA256	0x00000809
	TEE_TLS_RSA_PSS_PSS_SHA384	0x0000080A
	TEE_TLS_RSA_PSS_PSS_SHA512	0x0000080B
Legacy algorithms	TEE_TLS_RSA_PKCS_SHA1	0x00000201
	TEE_TLS_ECDSA_SHA1	0x00000203
Reserved Code Points	TEE_TLS_OBSOLETE_RESERVED	0x00000000 - 0x00000200
	TEE_TLS_DSA_SHA1_RESERVED	0x00000202
	TEE_TLS_OBSOLETE_RESERVED	0x00000204 - 0x00000400
	TEE_TLS_DSA_SHA256_RESERVED	0x00000402

Algorithm Group	Algorithm	Value
	TEE_TLS_OBSOLETE_RESERVED	0x00000404 - 0x00000500
	TEE_TLS_DSA_SHA384_RESERVED	0x00000502
	TEE_TLS_OBSOLETE_RESERVED	0x00000504 - 0x00000600
	TEE_TLS_DSA_SHA512_RESERVED	0x00000602
	TEE_TLS_OBSOLETE_RESERVED	0x00000604 - 0x000006FF
	TEE_TLS_PRIVATE_USE	0x0000FE00 - 0x0000FFFF
	Reserved for future use	All values not listed in the table are reserved for future use.
	TEE_TLS_SOCKET_SIGNATURE_SCHEME_ILLEGAL_VALUE	0xFFFFFFFF

TEE_TLS_SOCKET_SIGNATURE_SCHEME_ILLEGAL_VALUE is reserved for testing and validation and SHALL be treated as an undefined value when provided to the TEE_tlsSocket_Setup structure or the TEE_tlsSocket_SessionInfo structure.

C.2.6.5 TEE_tlsSocket_Tls13KeyExGroup

Since: Annex C TEE Sockets TLS API v1.1

```
typedef uint32_t TEE_tlsSocket_Tls13KeyExGroup;
```

The `TEE_tlsSocket_Tls13KeyExGroup` type provides values indicating the key exchange groups the TA supports for TLS 1.3 handshakes. Table C-10 defines the values of `TEE_tlsSocket_Tls13KeyExGroup`.

The TA must provide a priority-ordered array of these values. The TA must indicate the number of values in the array in the `numTls13KeyExGroups` variable. The array must contain at least one value. The array SHALL include only key exchange groups supported by the TEE ([TEE Core] Table 6-14). To determine whether the TEE supports a particular group, the TA can use the `TEE_IsAlgorithmSupported` function (see [TEE Core] section 6.2.9). If the list contains an algorithm the implementation does not support, the implementation SHALL return the `TLS_ISOCKET_TLS_ERROR_UNSUPPORTED_KEYEX_GROUP` error code.

The implementation will send the provided list to the server in the `supported_groups` extension of the `ClientHello` message. Note that the TA can use the `numTls13KeyShares` variable (see Table C-25) to control how many key shares are generated.

Table C-10: `TEE_tlsSocket_Tls13KeyExGroup` Values

Algorithm	Value	Main Reference
`TEE_TLS_KEYEX_GROUP_SECP256R1`	`0x00000017`	[RFC 4492]
`TEE_TLS_KEYEX_GROUP_SECP384R1`	`0x00000018`	
`TEE_TLS_KEYEX_GROUP_SECP521R1`	`0x00000019`	
`TEE_TLS_KEYEX_GROUP_X25519`	`0x0000001D`	
`TEE_TLS_KEYEX_GROUP_X448`	`0x0000001E`	
`TEE_TLS_KEYEX_GROUP_FFDHE_2048`	`0x00000100`	[RFC 7919]
`TEE_TLS_KEYEX_GROUP_FFDHE_3072`	`0x00000101`	
`TEE_TLS_KEYEX_GROUP_FFDHE_4096`	`0x00000102`	
`TEE_TLS_KEYEX_GROUP_FFDHE_6144`	`0x00000103`	
`TEE_TLS_KEYEX_GROUP_FFDHE_8192`	`0x00000104`	
Reserved by [RFC 8446]	`0x000001FC – 0x000001FF`	
Reserved by [RFC 8446]	`0x0000FE00 – 0x0000FEFF`	
Reserved for GlobalPlatform use	`0x0000FF00 – 0x0000FF0E`	
`TEE_TLS_KEYEX_GROUP_ILLEGAL_VALUE`	`0x0000FF0F`	
Reserved for implementation defined key exchange group	`0x0000FF10 – 0x0000FFFF`	

`TEE_TLS_KEYEX_GROUP_ILLEGAL_VALUE` is reserved for testing and validation and SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_Setup` structure or the `TEE_tlsSocket_SessionInfo` structure.

C.2.6.6 TEE_tlsSocket_PSK_Info Structure

```
typedef struct TEE_tlsSocket_PSK_Info_s {
    TEE_ObjectHandle    pskKey;
    char                *pskIdentity;
} TEE_tlsSocket_PSK_Info;
```

When PSK is used, the TA needs to provide the key and a key identity to the TLS implementation. This structure holds that information.

Table C-11: `TEE_tlsSocket_PSK_Info` Member Variables

Name	Purpose
`TEE_ObjectHandle pskKey`	An opened Persistent Object or an initialized Transient Object containing the PSK. The Object Type ([TEE Core] Table 6-13) must be `TEE_TYPE_GENERIC_SECRET` and the Object Attribute ([TEE Core] Table 6-15) must be `TEE_ATTR_SECRET_VALUE`.
`char *pskIdentity`	Pointer to a string containing the identity of the key. The interpretation of this string is something that the client and the server have agreed upon. The pointer MAY be NULL when the PSK is used for resumption in TLS 1.3 together with the associated ticket. The format must be a zero-terminated UTF-8 encoded string as defined in [TEE Core] section 3.2, Data Types.

C.2.6.7 TEE_tlsSocket_SessionTicket_Info Structure

Since: Annex C TEE Sockets TLS API v1.1

```
typedef struct TEE_tlsSocket_SessionTicket_Info_s {
    uint8_t                 *encrypted_ticket;
    uint32_t                encrypted_ticket_len;
    uint8_t                 *server_id;
    uint32_t                server_id_len;
    uint8_t                 *session_params;
    uint32_t                session_params_len;
    uint8_t                 caller_allocated;
    TEE_tlsSocket_PSK_Info  psk;
} TEE_tlsSocket_SessionTicket_Info;
```

When the implementation supports session ticket based resumption, the implementation SHALL use this structure to store a session ticket received from the server along with associated session information. The ticket may later be used for resumed TLS connections (resumed handshakes).

The implementation SHALL ensure that it follows the TLS specification regarding resumption. Especially, the implementation SHALL ensure that a resumed handshake uses the same protocol version, cipher suite, and `server_name` as the initial handshake. For this purpose, the implementation SHALL store the parameters of the initial session in the memory pointed to by `session_params`.

When the ticket is received in a TLS 1.3 connection, the resumption PSK associated with the ticket SHALL be stored in the `psk` field of `TEE_tlsSocket_SessionTicket_Info`.

When the ticket is received in a TLS 1.2 connection, the implementation SHALL store the master secret in `session_params`.

Memory management: When connecting to a server for the first time the TA MAY, if supporting resumption, provide an array of zeroed `TEE_tlsSocket_SessionTicket_Info` structures in the `TEE_tlsSocket_Setup` structure (section C.2.7). When the implementation receives a ticket from the server, the implementation SHALL locate the next unfilled structure in the provided array, if any. If an unfilled structure is found, the implementation SHALL allocate memory for storing the ticket, server ID, and session parameters. The implementation SHALL store the addresses of the allocated memory in the pointer fields of this structure and set the lengths appropriately. The TA may, after any point between a successful call to `open` and a call to `close`, take a deep copy of structure contents for its own storage. The implementation SHALL deallocate the memory pointed to by the structure when the connection is closed if the `caller_allocated` field is set to `0`. When the TA provides a filled ticket it wishes to use for resumption, it must set the `caller_allocated` field to 1.

484 **Table C-12: `TEE_tlsSocket_SessionTicket_Info` Member Variables**

Name	Purpose
`uint8_t *encrypted_ticket`	Pointer to memory where the implementation SHALL store the encrypted session ticket.
`uint32_t encrypted_ticket_len`	Length of the currently stored encrypted ticket.
`uint8_t *server_id`	Pointer to memory where the implementation SHALL store the identity of the server that sent the session ticket. If the TA sent the `server_name` extension, then the identity SHALL be the contents of that extension, i.e. the encoded `HostName` vector, defined in [RFC 6066], including the length octets. If the TA did not send the `server_name` extension, then the identity SHALL be the `subject` field of the server's certificate (see [RFC 5280]), i.e. the tag, length, and value of the DER-encoded ASN.1 `RDNSequence` type.
`uint32_t server_id_len`	Number of bytes pointed to by `server_id`.
`uint8_t *session_params`	Pointer to memory where the implementation SHALL store the parameters of the handshake when a ticket is received. The encoding and contents of the parameters are implementation defined. The implementation SHALL store enough session parameters to allow it later to check the prerequisites for session resumption mandated by the TLS specification, e.g. that the same cipher suite must be used in both the initial and the resumed connection.
`uint32_t session_params_len`	Number of bytes pointed to by `session_params`.
`uint8_t caller_allocated`	Specifies whether the memory pointed to by the `encrypted_ticket`, `server_id`, and `session_params` fields been allocated by the caller or the implementation. ▪ `0`: Allocated by the implementation ▪ `1`: Allocated by the caller ▪ `255`: Illegal value
`TEE_tlsSocket_PSK_Info psk`	If a ticket is received in a TLS 1.3 handshake, the implementation SHALL store the derived resumption PSK here.

485

C.2.6.8 TEE_tlsSocket_SRP_Info Structure

```
typedef struct TEE_tlsSocket_SRP_Info_s {
    char *srpPassword;
    char *srpIdentity;
} TEE_tlsSocket_SRP_Info;
```

When SRP is used, the TA needs to provide the password and the user identity to the TLS implementation. This structure holds that information. Note that SRP is supported in TLS 1.2 and earlier versions, but not in TLS 1.3.

Table C-13: `TEE_tlsSocket_SRP_Info` Member Variables

Name	Purpose
`char *srpPassword`	Pointer to the password. The format must be a zero-terminated UTF-8 encoded string as defined in [TEE Core] section 3.2, Data Types.
`char *srpIdentity`	Pointer to the user name or identity corresponding to the password. The format must be a zero-terminated UTF-8 encoded string as defined in [TEE Core] section 3.2, Data Types.

C.2.6.9 TEE_tlsSocket_ClientPDC Structure

Since: Annex C TEE Sockets TLS API v1.1 – See Backward Compatibility note below.

```
typedef struct TEE_tlsSocket_ClientPDC_s {
    TEE_ObjectHandle    privateKey;
    uint8_t             *bulkCertChain;
    uint32_t            bulkSize;
    // The following field was introduced in v1.1
    uint32_t            bulkEncoding;
} TEE_tlsSocket_ClientPDC;
```

This structure holds a handle to the private key and a certificate chain that the implementation (i.e. the client) SHALL use to authenticate or attest itself during the TLS handshake.

Memory management: The memory pointed to by `bulkCertChain` SHALL be fully managed by the TA.

Table C-14: `TEE_tlsSocket_ClientPDC` Member Variables

Name	Purpose
`TEE_ObjectHandle privateKey`	An opened Persistent Object or initialized Transient Object containing the private key corresponding to the public key in the certificate.
`uint8_t *bulkCertChain`	Pointer to the client's certificate chain. The certificates must be in child-to-parent order, i.e. the client's end-entity certificate must be first. The end-entity certificate must contain the public key corresponding to `privateKey`.
`uint32_t bulkSize`	The size of `*bulkCertChain`.
`uint32_t bulkEncoding`	A bit mask that indicates the format(s) in which certificates in `*bulkCertChain` are encoded: `0x00000001` – X.509 DER `0x00000002` – X.509 PEM `0x80000000` – Illegal bit setting `0x7F000000` – Bits reserved for implementation All other bits are reserved by GlobalPlatform. When multiple bits are set, the certificates may be in any of the enabled formats. In this case, the implementation SHALL detect the format of the certificate, e.g. by trial-and-error parsing. The implementation SHALL support X.509 DER encoding.

`bulkEncoding = 0x80000000` is reserved for testing and validation and SHALL be treated as an undefined value when provided in the `TEE_tlsSocket_Credentials` structure.

Backward Compatibility

Prior to Annex C TEE Sockets TLS API v1.1, `char*` was used as the type for `bulkCertChain`.

The `bulkEncoding` field was introduced in Annex C TEE Sockets TLS API v1.1.

517

518 C.2.6.10 TEE_tlsSocket_ServerCredentialType

519 **Since:** Annex C TEE Sockets TLS API v1.1 – See Backward Compatibility note below.

520
```
typedef uint32_t TEE_tlsSocket_ServerCredentialType;
```

521

522 The `TEE_tlsSocket_ServerCredentialType` type indicates how the client shall authenticate the server.
523 Table C-15 defines the values of `TEE_tlsSocket_ServerCredentialType`.

524 **Note:** `TEE_tlsSocket_ServerCredentialType` does not have a `TEE_TLS_PEER_CRED_NONE` member
525 due to security risks associated with not validating remote endpoints.

526 **Table C-15: `TEE_tlsSocket_ServerCredentialType` Values**

Name	Value	Meaning
`TEE_TLS_SERVER_CRED_PDC`	`0x00000000`	Legacy option, where the client has the server's public key and will use it to decrypt and verify messages during the handshake. When this option is used, the certificate chain received from the server is ignored. For backward compatibility; not recommended for new applications.
`TEE_TLS_SERVER_CRED_CSC`	`0x00000001`	The client has at least one trusted certificate that will be used to validate the server's certificate chain.
`TEE_TLS_SERVER_CRED_CERT_PIN`	`0x00000002`	Server SHALL be authenticated based on whether the SHA-256 hash of the server's certificate matches one of the pinned values.
`TEE_TLS_SERVER_CRED_PUBKEY_PIN`	`0x00000003`	Server SHALL be authenticated based on whether the SHA-256 hash of the `SubjectPublicKeyInfo` structure in the server's certificate matches one of the pinned values.
Reserved for GlobalPlatform use	`0x00000004` – `0x7FFFFFFE`	Reserved by GlobalPlatform for future use.
`TEE_TLS_SERVER_CRED_ILLEGAL_VALUE`	`0x7FFFFFFF`	Reserved for testing and validation and SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_Credentials` structure.
Implementation defined	`0x80000000` – `0xFFFFFFFF`	Reserved for proprietary use.

527

Backward Compatibility

Prior to Annex C TEE Sockets TLS API v1.1, `TEE_tlsSocket_ServerCredentialType` was defined as an `enum`.

In Annex C TEE Sockets TLS API v1.1, the `TEE_TLS_SERVER_CRED_PDC` value became a legacy option recommended only for backward compatibility.

C.2.6.10.1 Server Certificate Chain Validation

When the TA has chosen the `TEE_TLS_SERVER_CRED_CSC` server credential type, the implementation SHALL perform certification path validation according to [RFC 5280] for the server's certificate chain it receives during the handshake. Implementing the full validation process specified by [RFC 5280] may require a large amount of code, however, so this document specifies the following validation steps that the implementation SHALL perform, at minimum:

- The `subject` field or the `subjectAltName` extension in the child-most certificate matches the `server_name` provided by the TA.
- The public key in each certificate, except the child-most certificate, successfully verifies the signature of the preceding certificate.
- For each certificate except the child-most, the `cA` bit in the `basicConstraints` extension is set.
- The path length constraint included in the `basicConstraints` extension is not exceeded.
- The `keyUsage` extension of each certificate, except the child-most certificate, allows certificate signing (i.e. has the `keyCertSign` bit set).
- The extended `keyUsage` extension of the child-most certificate allows TLS server authentication (i.e. contains the `id-kp-serverAuth` object identifier).
- For TLS 1.2 and earlier handshakes, the `keyUsage` extension of the child-most certificate allows the authentication method used in the handshake: `digitalSignature` or `keyEncipherment`. Because TLS 1.3 only supports signature-based authentication when certificates are used, in TLS 1.3 handshakes the `keyUsage` extension SHALL have the `digitalSignature` bit set.
- If revocation information is available, e.g. because a CRL distribution point or the URL of an OCSP responder was listed in the issuer certificate, or when the server sent a stapled OCSP response, then the implementation SHALL perform the revocation check and each certificate SHALL have non-revoked status.
- For each certificate, the current date is between the `notBefore` and `notAfter` dates of the certificate. This check SHALL be performed when either of the following is true:
  1) The `gpd.tee.systemTime.protectionLevel` property (defined in [TEE Core]) has the value `1000`, or
  2) The TA has set the `allowTAPersistentTimeCheck` field in the server credentials structure to a non-zero value.

  Two options are then available:
  a) In the former case (1), the implementation SHALL retrieve the current time using the `TEE_GetSystemTime` function.
  b) In the latter case (2), the implementation SHALL retrieve the current time using the `TEE_GetTAPersistentTime` function.

  If both methods are available, then option (b) SHALL take priority.

The implementation SHOULD implement further validation steps from [RFC 5280]. These may include, for example, `nameConstraints` or certificate policy checks.

The TA can use the `gpd.tee.tls.auth.remote.validation_steps` property to determine which validation steps are supported by the implementation. The value of the property is a `uint32_t`. Table C-16 defines the bit-mask constants for `gpd.tee.tls.auth.remote.validation_steps`.

Table C-16: `gpd.tee.tls.auth.remote.validation_steps` Property Bit-mask Constants

Name	Value
`TEE_TLS_AUTH_REMOTE_VALIDATION_STEP_NAME_CONSTRAINTS`	`0x00000001`
`TEE_TLS_AUTH_REMOTE_VALIDATION_STEP_POLICY_CONSTRAINTS`	`0x00000002`
Reserved for GlobalPlatform use	`0x007FFFE0`
`TEE_TLS_AUTH_REMOTE_VALIDATION_STEP_ILLEGAL_VALUE`	`0x00800000`
Implementation defined	`0xFF000000`

`TEE_TLS_AUTH_REMOTE_VALIDATION_STEP_ILLEGAL_VALUE` is reserved for testing and validation and SHALL be treated as an undefined value when the corresponding bit is set in the value retrieved as the `gpd.tee.tls.auth.remote.validation_steps` property.

C.2.6.11 TEE_tlsSocket_ServerPDC Structure

Since: Annex C TEE Sockets TLS API v1.1 – See Backward Compatibility note below.

```
typedef struct TEE_tlsSocket_ServerPDC_s {
    TEE_ObjectHandle    publicKey;
    // The following fields were introduced in v1.1
    TEE_ObjectHandle    *trustedCerts;
    uint32_t            *trustedCertEncodings;
    uint32_t            numTrustedCerts;
    uint32_t            allowTAPersistentTimeCheck;
    uint8_t             *certPins;
    uint32_t            numCertPins;
    uint8_t             *pubkeyPins;
    uint32_t            numPubkeyPins;
} TEE_tlsSocket_ServerPDC;
```

This structure holds the credentials the client will use to authenticate the server or verify the server's attestation evidence.

Table C-17: `TEE_tlsSocket_ServerPDC` Member Variables

Name	Purpose
`TEE_ObjectHandle publicKey`	Handle of the server's public key. See the description of `TEE_TLS_SERVER_CRED_PDC` in Table C-15. This option is for backward compatibility and not recommended for new applications.
`TEE_ObjectHandle *trustedCerts`	Pointer to an array of one or more object handles, where each object contains one or more trusted certificates. The trusted certificates are used in the validation of the server's certificate chain. See the description of `TEE_TLS_SERVER_CRED_CSC` in Table C-15 for more information.
`uint32_t *trustedCertEncodings`	Pointer to an array of bit masks that indicate the format in which the certificates in each object in `trustedCerts` are encoded. The possible values are: `0x00000001` X.509 DER `0x00000002` X.509 PEM `0x80000000` Illegal bit setting (See note following table.) `0x7F000000` Bits reserved for implementation All other bits are reserved by GlobalPlatform. When multiple bits are set, the certificates may be in any of the enabled formats. In this case, the implementation SHALL detect the format of the certificate, e.g. by trial-and-error parsing. The implementation SHALL support X.509 DER encoding.
`uint32_t numTrustedCerts`	The number of object handles in `trustedCerts`.

Name	Purpose
`uint32_t allowTAPersistentTimeCheck`	An option that indicates whether the implementation is allowed to retrieve the current time using the `TEE_GetTAPersistentTime` when validating the `notBefore` and `notAfter` dates in the server's certificate chain. Note that the restrictions in section C.2.6.10.1 apply. The possible values are: `0` — Not allowed `1` — Allowed `0xFFFFFFFF` — Illegal value (See note following table.)
`uint8_t *certPins`	Pointer to SHA-256 hashes of trusted certificates. See the description of `TEE_TLS_SERVER_CRED_CERT_PIN` in Table C-15.
`uint32_t numCertPins`	Number of hashes in `certPins`.
`uint8_t *pubkeyPins`	Pointer to SHA-256 hashes of trusted public key `SubjectPublicKeyInfo` structures. See the description of `TEE_TLS_SERVER_CRED_PUBKEY_PIN` in Table C-15.
`uint32_t numPubkeyPins`	Number of hashes in `pubkeyPins`.

`trustedCertEncodings = 0x80000000` and `allowTAPersistentTimeCheck = 0xFFFFFFFF` are reserved for testing and validation and each SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_Credentials` structure.

Backward Compatibility

The fields below `publicKey` were added in Annex C TEE Sockets TLS API v1.1.

In Annex C TEE Sockets TLS API v1.1, the `publicKey` field became a legacy option recommended only for backward compatibility.

608 ### C.2.6.12 TEE_tlsSocket_ClientCredentialType

609 **Since:** Annex C TEE Sockets TLS API v1.1 – See Backward Compatibility note below.

610
```
typedef uint32_t TEE_tlsSocket_ClientCredentialType;
```

611

612 The `TEE_tlsSocket_ClientCredentialType` type indicates the type of credentials the TA has.
613 Table C-18 defines the values of `TEE_tlsSocket_ClientCredentialType`.

614 **Table C-18: `TEE_tlsSocket_ClientCredentialType` Values**

Name	Value	Meaning
`TEE_TLS_CLIENT_CRED_NONE`	`0x00000000`	TA has no credentials.
`TEE_TLS_CLIENT_CRED_PDC`	`0x00000001`	TA has pre-distributed credentials; i.e. a PSK or an SRP password.
`TEE_TLS_CLIENT_CRED_CSC`	`0x00000002`	TA has certificate storage credentials; i.e. a private key and a certificate.
Reserved for GlobalPlatform use	`0x00000003` - `0x7FFFFFFE`	Reserved by GlobalPlatform for future use.
`TEE_TLS_CLIENT_CRED_ILLEGAL_VALUE`	`0x7FFFFFFF`	Reserved for testing and validation and SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_Credentials` structure.
Implementation defined	`0x80000000` - `0xFFFFFFFF`	Reserved for proprietary use.

615 #### Backward Compatibility

616 Prior to Annex C TEE Sockets TLS API v1.1, `TEE_tlsSocket_ClientCredentialType` was defined as
617 an `enum`.

618

C.2.6.13 TEE_tlsSocket_Credentials Structure

Since: Annex C TEE Sockets TLS API v1.1

```
typedef struct TEE_tlsSocket_Credentials_s {
    TEE_tlsSocket_ServerCredentialType  serverCredType;
    TEE_tlsSocket_ServerPDC             *serverCred;
    TEE_tlsSocket_ClientCredentialType  clientCredType;
    TEE_tlsSocket_ClientPDC             *clientCred;
} TEE_tlsSocket_Credentials;
```

This structure contains information on what kind of credentials the TA holds for itself and for the server.

This structure is used to specify credentials for both endpoint authentication and remote attestation.

Table C-19: `TEE_tlsSocket_Credentials` Member Variables

Name	Purpose
`TEE_tlsSocket_ServerCredentialType serverCredType`	The provided server credential type. See Table C-15 for possible values.
`TEE_tlsSocket_ServerPDC *serverCred`	Pointer to the provided server credentials used to authenticate the server or verify the server's attestation evidence.
`TEE_tlsSocket_ClientCredentialType clientCredType`	The provided client credential type. See Table C-18 for possible values.
`TEE_tlsSocket_ClientPDC *clientCred`	Pointer to the provided credentials the client uses to authenticate or attest itself to the server.

Note: Implementations may define additional credential types.

C.2.6.14 TEE_tlsSocket_CB_Data Structure

```
typedef struct TEE_tlsSocket_CB_Data_s {
    uint32_t  cb_data_size;
    uint8_t   cb_data[];
} TEE_tlsSocket_CB_Data;
```

This structure is returned in the output buffer by the `ioctl` function `TEE_TLS_BINDING_INFO`.

For TLS 1.2 connections, it provides `tls-unique` channel bindings according to [RFC 5929].

For TLS 1.3 connections, it provides the value `TLS-Exporter(label, context_value, key_length)` according to [RFC 8446], where `label` is the caller-provided value contained in the `buf` argument provided to the `ioctl` call and used to indicate the use case of the channel binding information, `context_value` is empty, and `key_length` is 32. The input secret used in the computation of the exporter value SHALL be the exporter master secret of the connection.

Table C-20: `TEE_tlsSocket_CB_Data` Member Variables

Name	Purpose
`uint32_t cb_data_size`	The size of the channel binding data in `cb_data[]`.
`uint8_t cb_data[]`	The channel binding data.

Memory management note: The implementation SHALL store the channel binding data in the output buffer provided by the TA in the `ioctl` call.

C.2.6.15 TEE_tlsSocket_SessionInfo Structure

Since: Annex C TEE Sockets TLS API v1.1

```
typedef struct TEE_tlsSocket_SessionInfo_s
{
    uint8_t                         structVersion;
    TEE_tlsSocket_TlsVersion        chosenVersion;
    uint32_t                        chosenCiphersuite;
    TEE_tlsSocket_SignatureScheme   chosenSigAlg;
    TEE_tlsSocket_Tls13KeyExGroup   chosenKeyExGroup;
    unsigned char                   *matchedServerName;
    uint32_t                        matchedServerNameLen;
    const uint8_t                   *validatedServerCertificate;
    uint32_t                        validatedServerCertificateLen;
    uint32_t                        usedServerAuthenticationMethod;
    /* The following was added in v1.2: */
    TEE_tlsSocket_AttEvTransMethod  usedServerAttestationMethod;
} TEE_tlsSocket_SessionInfo;
```

This structure is returned in the output buffer by the `ioctl` function `TEE_TLS_SESSION_INFO`.

The contents of the structure can be used by the TA to discover session information for the current TLS session.

Table C-21: `TEE_tlsSocket_SessionInfo` Member Variables

Name	Purpose
`uint8_t structVersion`	Version number of this structure type. The possible values include: `0` — The previous version defined in TEE Sockets API Annex C TLS v1.1. `1` — The current version defined in this specification `255` — Illegal value (See note following table.)
`TEE_tlsSocket_TlsVersion chosenVersion`	The negotiated TLS protocol version used in this session
`uint32_t chosenCiphersuite`	The negotiated cipher suite used in this session
`TEE_tlsSocket_SignatureScheme chosenSigAlg`	The negotiated signature algorithm that was used to authenticate the server during the handshake
`TEE_tlsSocket_Tls13KeyExGroup chosenKeyExGroup`	The negotiated key exchange group used in this session
`unsigned char* matchedServerName`	Pointer to memory storing the server name provided by the TA in the session options that matched the server identity.
`uint32_t matchedServerNameLen`	Number of bytes pointed to by `matchedServerName`. The length SHALL be set to `0` if the handshake did not use certificate-based server authentication.

<table>
<tr><th>Name</th><th>Purpose</th></tr>
<tr><td>const uint8_t *validatedServerCertificate</td><td>Pointer to memory where the implementation has stored the successfully validated server certificate chain. The chain SHALL be stored by concatenating the DER encodings of the certificates, in child-to-parent order.
The pointed memory SHALL be considered valid only if all of the following conditions are fulfilled:
• Certificate-based server authentication method was used in the TLS handshake.
• The TA had enabled the storeServerCertChain option in the session options.
• The TEE_TLS_RELEASE_CERT_CHAIN ioctl command has not been invoked for the connection.
This option can be used by the TA to e.g. extend the implementation's certificate chain validation with custom validation steps. In such a use case, the TA is responsible for examining the certificate chain according to the TA's policy and for terminating the TLS connection in case of validation failure.</td></tr>
<tr><td>uint32_t validatedServerCertificateLen</td><td>Length of the stored server certificate chain. The length SHALL be set to 0 if no certificate chain is available.</td></tr>
<tr><td>uint32_t usedServerAuthenticationMethod</td><td>Indicates the server authentication method used in the TLS handshake. Possible values are:<table><tr><td>0</td><td>Server's certificate chain was validated against the provided trust root certificates</td></tr><tr><td>1</td><td>Server's certificate chain was validated against the provided trusted certificate pins</td></tr><tr><td>2</td><td>Server was authenticated using a PSK</td></tr><tr><td>3</td><td>Server was authenticated using SRP</td></tr><tr><td>0xFFFFFFFF</td><td>Illegal value (See note following table.)</td></tr></table></td></tr>
<tr><td>TEE_tlsSocket_AttEvTransMethod usedServerAttestationMethod</td><td>Indicates the attestation evidence transmission method the server used to send attestation evidence during the handshake. If no attestation evidence was received from the server, the value is set to TEE_TLS_ATT_EV_TRANS_METHOD_NONE.</td></tr>
</table>

674

675 structVersion = 255 and usedServerAuthenticationMethod = 255 are reserved for testing and
676 validation and each SHALL be treated as an undefined value when retrieved as TEE_TLS_SESSION_INFO.

677 **Memory management note:** The implementation SHALL store the matchedServerName and
678 validatedServerCertificate in the output buffer provided by the TA in the ioctl call.

C.2.6.16 TEE_tlsSocket_AttFlags

Since: Annex C TEE Sockets TLS API v1.2

This bit mask variable configures the use of remote attestation in the TLS handshake. The following bit flags are supported:

Table C-22: `TEE_tlsSocket_AttFlags` Values

Name	Value	Meaning
`TEE_TLS_ATT_FLAG_SEND_EVIDENCE`	`0x00000001`	The implementation SHALL send evidence when requested by the remote endpoint.
`TEE_TLS_ATT_FLAG_SEND_UNSOLICITED_EVIDENCE`	`0x00000002`	The implementation SHALL send evidence even when no evidence is requested by the remote endpoint. • This option may cause the remote endpoint to abort the handshake if the `TEE_TLS_ATT_EV_TRANS_METHOD_CERT_MSG_EXT` evidence transmission method is used, since the TLS specification requires aborting the handshake when unsolicited TLS extensions are received. • The `TEE_TLS_ATT_EV_TRANS_METHOD_X509_EXTENSION` and `TEE_TLS_ATT_EV_TRANS_METHOD_EXTRA_CERT` transmission methods will not violate the TLS specification when used together with this option.
`TEE_TLS_ATT_FLAG_REQUEST_EVIDENCE`	`0x00000004`	The implementation SHALL request evidence from the remote endpoint.
`TEE_TLS_ATT_FLAG_REQUIRE_EVIDENCE`	`0x00000008`	The implementation SHALL terminate the handshake if any of the following occur: • No evidence is received from the remote endpoint. • The implementation cannot verify the evidence signature using the provided verification trust anchor. • The channel bindings in the evidence do not match the channel bindings value the implementation independently computed based on the current handshake.

Name	Value	Meaning
TEE_TLS_ATT_FLAG_PRIVACY	0x00000010	The implementation SHALL NOT include privacy-sensitive claims in the attestation evidence. It is up to the implementation to define (and document) which claims are deemed privacy-sensitive.
TEE_TLS_ATT_FLAG_USE_ATTESTATION_SERVICE	0x00000020	The implementation SHALL use an attestation service identified by the AttEnvUUID field in the TEE_tlsSocket_AttestationSetup structure. (Note that if AttEnvUUID is NULL, then the implementation SHALL use the default attestation service.)
Reserved for GlobalPlatform use	0x00000040 – 0x7FFFFFFE	Reserved by GlobalPlatform for future use.
TEE_TLS_ATT_FLAG_ILLEGAL_VALUE	0x7FFFFFFF	Reserved for testing and validation and SHALL be treated as an undefined value when provided to the TEE_tlsSocket_AttestationSetup structure.
Implementation defined	0x80000000 – 0xFFFFFFFF	Reserved for proprietary use.

684

C.2.6.17 TEE_tlsSocket_AttEvTransMethod

Since: Annex C TEE Sockets TLS API v1.2

Variables of this type indicate attestation evidence transmission methods. The type is used in the `TEE_tlsSocket_AttestationSetup` structure to indicate the evidence transmission method the TA shall use, as well as the accepted evidence transmission methods the server is allowed to use.

Table C-23: `TEE_tlsSocket_AttEvTransMethod` Values

Name	Value	Meaning
`TEE_TLS_ATT_EV_TRANS_METHOD_NONE`	`0x00000000`	No attestation evidence transmission methods are to be used or supported.
`TEE_TLS_ATT_EV_TRANS_METHOD_X509_EXTENSION`	`0x00000001`	Attestation evidence is transmitted in an X.509 v3 extension in the leaf certificate of the endpoint authentication certificate chain.
`TEE_TLS_ATT_EV_TRANS_METHOD_EXTRA_CERT`	`0x00000002`	Attestation evidence is transmitted in an extra certificate appended to the endpoint authentication certificate chain.
`TEE_TLS_ATT_EV_TRANS_METHOD_CERT_MSG_EXT`	`0x00000004`	Attestation evidence is transmitted in a TLS extension in the Certificate handshake message. This method SHALL be used only in TLS 1.3 handshakes.
Reserved for GlobalPlatform use	`0x00000005` – `0x7FFFFFFE`	Reserved by GlobalPlatform for future use.
`TEE_TLS_ATT_EV_TRANS_METHOD_ILLEGAL_VALUE`	`0x7FFFFFFF`	Reserved for testing and validation and SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_AttestationSetup` structure.
Implementation defined	`0x80000000` – `0xFFFFFFFF`	Reserved for proprietary use.

C.2.6.18 TEE_tlsSocket_AttestationSetup Structure

Since: Annex C TEE Sockets TLS API v1.2

This structure configures whether and how remote attestation shall be performed in the TLS handshake.

Note that if the flags variable is all zero, the attestation feature is disabled and evidence shall be neither transmitted nor requested.

```
typedef struct TEE_tlsSocket_AttestationSetup_s {
  TEE_tlsSocket_AttFlags          flags;
  TEE_tlsSocket_AttEvTransMethod  sendEvTransMethod;
  TEE_tlsSocket_AttEvTransMethod  recvEvTransMethod;
  TEE_tlsSocket_Credentials       *evidenceCred;
  TEE_UUID                        *attEnvUUID;
} TEE_tlsSocket_AttestationSetup;
```

Table C-24: `TEE_tlsSocket_AttestationSetup` Member Variables

Name	Purpose
`TEE_tlsSocket_AttFlags flags`	Bit flags that indicate whether and how remote attestation should be performed during the TLS handshake. If all flags are `0`: • Attestation will not be used (attestation evidence will be neither transmitted nor requested from the remote endpoint). • The rest of the fields in the attestation setup structure SHALL be ignored by the implementation.
`TEE_tlsSocket_AttEvTransMethod sendEvTransMethod`	Attestation evidence transmission method. Note that multiple bits may be set in the bit flag variable, indicating multiple supported methods. The implementation SHALL transmit evidence according to the method specification. If multiple methods are enabled, then the implementation shall pick one that the server has indicated support for.
`TEE_tlsSocket_AttEvTransMethod recvEvTransMethod`	Supported evidence reception methods. Note that multiple bits may be set in the bit flag variable, indicating multiple supported methods. If the evidence is received using a method that is not specified in this variable, the implementation SHALL abort the handshake.

Name	Purpose
`TEE_tlsSocket_Credentials *evidenceCred`	Evidence protection and verification credentials. The implementation SHALL use the client credentials (if not NULL) to protect the evidence it transmits during the handshake. The implementation SHALL use the server credentials (if not NULL) to verify the evidence it receives during the handshake. If verification of the evidence using the provided credentials fails, the implementation SHALL abort the handshake and return an error. If no client credentials are provided, the implementation SHALL either return an error, or use any suitable evidence protection credentials to protect evidence. If no server credentials are provided, the implementation SHALL either return an error or use any suitable evidence verification credentials to verify evidence.
`TEE_UUID *attEnvUUID`	UUID of the entity that the TA wants to use as the attesting environment (i.e. as the entity that generates and signs attestation evidence). This could be, for example, the UUID of a TA that provides an attestation service. If the implementation is unable to connect to the service with the given UUID, or if the service cannot generate evidence, the implementation SHALL return an error. If the pointer is NULL, then the implementation SHALL use the default attestation service.

707

C.2.7 TEE_tlsSocket_Setup Structure

Since: Annex C TEE Sockets TLS API v1.1

The setup structure is used to pass initialization information to the `open` function. An implementation MAY add proprietary variables to this structure to enable specific features, but for all conformant implementations, the `TEE_tlsSocket_Setup` structure SHALL include the following:

```
typedef struct TEE_tlsSocket_Setup_s {
    TEE_tlsSocket_TlsVersion acceptServerVersion;
    TEE_tlsSocket_CipherSuites_GroupA *allowedCipherSuitesGroupA;
    TEE_tlsSocket_PSK_Info *PSKInfo;
    TEE_tlsSocket_SRP_Info *SRPInfo;
    TEE_tlsSocket_Credentials *credentials;
    TEE_iSocket *baseSocket;
    TEE_iSocketHandle *baseContext;

    // The following fields were introduced in v1.1
    TEE_tlsSocket_CipherSuites_GroupB *allowedCipherSuitesGroupB;
    TEE_tlsSocket_SignatureScheme *sigAlgs;
    uint32_t numSigAlgs;
    TEE_tlsSocket_SignatureScheme *certSigAlgs;
    uint32_t numCertSigAlgs;
    TEE_tlsSocket_Tls13KeyExGroup *tls13KeyExGroups;
    uint32_t numTls13KeyExGroups;
    uint32_t numTls13KeyShares;
    TEE_tlsSocket_SessionTicket_Info  *sessionTickets;
    uint32_t sessionTicketsNumElements;
    uint32_t numStoredSessionTickets;
    unsigned char *serverName;
    uint32_t serverNameLen;
    uint8_t *serverCertChainBuf;
    uint32_t *serverCertChainBufLen;
    uint8_t storeServerCertChain;
    unsigned char **alpnProtocolIds;
    uint32_t *alpnProtocolIdLens;
    uint32_t numAlpnProtocolIds;

    // The following fields were introduced in v1.2
    TEE_tlsSocket_AttestationSetup *attestationSetup;
} TEE_tlsSocket_Setup;
```

748 **Table C-25: `TEE_tlsSocket_Setup` Member Variables**

Name	Purpose
`TEE_tlsSocket_TlsVersion acceptServerVersion`	Which version of the TLS protocol to accept from the server.
`TEE_tlsSocket_CipherSuites_GroupA *allowedCipherSuitesGroupA`	Pointer to an array of the TLS 1.2 cipher suites that the client offers to the server. The array is terminated with the value `TEE_TLS_NULL_WITH_NULL_NULL`. Note that the implementation SHALL NOT support this cipher suite. It is only used to terminate the list.
`TEE_tlsSocket_PSK_Info *PSKInfo`	Pointer to a structure holding the information for a PSK session.
`TEE_tlsSocket_SRP_Info *SRPInfo`	Pointer to a structure holding the information for an SRP session.
`TEE_tlsSocket_Credentials *credentials`	Pointer to a structure holding credential information.
`TEE_iSocket *baseSocket`	Pointer to the lower layer `TEE_iSocket` protocol. The lower layer protocol must be connection-oriented and reliable. A TCP socket is allowed, but a UDP socket is not.
`TEE_iSocketHandle *baseContext`	Pointer to the handle of the lower layer instance.
`TEE_tlsSocket_CipherSuites_GroupB *allowedCipherSuitesGroupB`	Pointer to an array of the TLS 1.3 cipher suites that the client offers to the server. The array is terminated with the value `TEE_TLS_NULL_WITH_NULL_NULL`. Note that the implementation SHALL NOT support this cipher suite. It is only used to terminate the list. When cipher suites for both TLS 1.3 and below are included, the implementation SHALL list the TLS 1.3 cipher suites first (with higher priority) in the `ClientHello` message.
`TEE_tlsSocket_SignatureScheme *sigAlgs`	Pointer to an array of signature algorithms the client supports for `CertificateVerify` handshake message signature verification. The array SHALL be in priority order (highest to lowest).
`uint32_t numSigAlgs`	The number of signature algorithms in the `sigAlgs` array.
`TEE_tlsSocket_SignatureScheme *certSigAlgs`	Pointer to an array of signature algorithms the client supports for certificate signature authentication in TLS 1.3 connections. The array SHALL be in priority order (highest to lowest). The array may be empty when TLS 1.3 has not been enabled by the TA.
`uint32_t numCertSigAlgs`	The number of signature algorithms in the `certSigAlgs` array.
`TEE_tlsSocket_Tls13KeyExGroup *tls13KeyExGroups`	Pointer to an array of key exchange groups the client offers to the server for TLS 1.3 connections. The array SHALL be in priority order (highest to lowest).

Name	Purpose
`uint32_t numTls13KeyExGroups`	The number of key groups in the `tls13KeyExGroups` array.
`uint32_t numTls13KeyShares`	Number of key shares the client shall offer for TLS 1.3 connections. The implementation SHALL generate `numTls13KeyShares` shares for the groups listed in `tls13KeyExGroups`, starting from the group at index `0`. If `numTls13KeyShares` is `0`, but the TA has enabled TLS 1.3, then the implementation SHALL offer a single key share for the highest-priority group in `tls13KeyExGroups`.
`TEE_tlsSocket_SessionTicket_Info *sessionTickets`	Pointer to an array of structures in which the implementation SHALL store received session tickets.
`uint32_t sessionTicketsNumElements`	Number of elements in the `sessionTickets` array.
`uint32_t numStoredSessionTickets`	Number of session tickets stored in the `sessionTickets` array, i.e. the first `numStoredSessionTickets` elements of `sessionTickets` are currently filled.
`unsigned char *serverName`	Pointer to the name of the server the TA wants to connect to, encoded according to [RFC 6066] section 3. The implementation SHALL send the value in the `HostName` field of the `server_name` extension defined in [RFC 6066] section 3. When using certificate-based server authentication, the implementation SHALL compare the name to the identity in the server's certificate, as described in section C.2.6.10.1.
`uint32_t serverNameLen`	Number of bytes pointed to by `serverName`.
`uint8_t *serverCertChainBuf`	Pointer to memory where the implementation SHALL store the server's certificate chain received during the TLS handshake. The pointed memory SHALL be considered valid even when the TLS handshake was unsuccessful, as long as the implementation received the complete server `Certificate` message, making this mechanism useful for debugging. The TA should examine the error code to determine whether the `Certificate` message was successfully received in a failed TLS handshake. The TA may set the value to NULL, in which case the implementation SHALL NOT store the server certificate chain for failed TLS handshakes.
`uint32_t *serverCertChainBufLen`	Pointer to length of the `serverCertChainBuf` buffer. The implementation SHALL store the length of the stored certificate chain in the pointed variable.

Name	Purpose
`uint8_t storeServerCertChain`	This option specifies whether the implementation should store the received server certificate chain when a TLS session is successfully established. Possible values are: `0` — Do not store the server's certificate chain (e.g. release the chain immediately after the implementation has validated it). 1 — Store the server's certificate chain such that the `TEE_TLS_SESSION_INFO ioctl` command can be used to retrieve a pointer to memory holding the server's certificate chain. (See [TEE Sockets] section 5.2.9 for `ioctl` details). 255 — Illegal Value (See note following table.) As an optimization, when both `storeServerCertChain` is set to 1 and `serverCertChainBuf` is not set to `NULL`, the implementation MAY use the memory pointed to by `serverCertChainBuf` to store the server certificate chain even for successful connections. In this case, the pointer returned by the `TEE_TLS_SESSION_INFO` command will point to the same memory as `serverCertChainBuf`.
`unsigned char **alpnProtocolIds`	An array of pointers to IANA-registered ALPN protocol identification sequences. The implementation SHALL transmit these in the ALPN `ClientHello` extension as specified in [RFC 7301].
`uint32_t *alpnProtocolIdLens`	Length (number of bytes) of each protocol identification sequence pointed to by `alpnProtocolIds`.
`uint32_t numAlpnProtocolIds`	Number of protocol identification sequences pointed to by `alpnProtocolIds`.
`TEE_tlsSocket_AttestationSetup *attestationSetup`	Remote attestation configuration. If NULL, remote attestation SHALL NOT be used in the handshake.

`storeServerCertChain = 255` is reserved for testing and validation and SHALL be treated as an undefined value when provided to the `TEE_tlsSocket_Setup` structure.

Memory management note: As stated in [TEE Sockets] section 5.2.4, after `open` has been successfully called, "any changes to the `setup` parameter SHALL NOT alter the behavior of the protocol in subsequent calls to the instance `TEE_iSocket` functions". One way the implementation could fulfill this requirement is to take a deep copy of the `TEE_tlsSocket_Setup` structure and use the copy instead of the original.

Examples of how to configure the setup structure are given in Annex D ([Sockets Examples]) sections D.2 and D.3.

759

760 ### C.2.8 Instance Specific Errors

761 **Table C-26: TLS Instance Specific Errors**

Name	Value	Function	Fatal	Meaning
TEE_ISOCKET_TLS_ERROR_REJECTED_SUITE	0xF1030001	open	Yes	The server rejected all the offered cipher suites.
TEE_ISOCKET_TLS_ERROR_VERSION	0xF1030002	open	Yes	The server does not support the TLS version(s) provided by this implementation.
TEE_ISOCKET_TLS_ERROR_UNSUPPORTED_SUITE	0xF1030003	open	Yes	The combination of algorithms (authentication and key exchange, encryption, and message authentication) is not supported.
TEE_ISOCKET_TLS_ERROR_HANDSHAKE	0xF1030004	open	Yes	An error occurred during the TLS handshake.
TEE_ISOCKET_TLS_ERROR_AUTHENTICATION	0xF1030005	open	Yes	The server could not be authenticated.
TEE_ISOCKET_TLS_ERROR_DATA	0xF1030006	close	Yes	Invalid data was received (incorrect authentication value or other protocol error).
TEE_ISOCKET_TLS_ERROR_UNSUPPORTED_KEYEX_GROUP	0xF1030007	open	Yes	The implementation does not support all the selected key exchange groups.
TEE_ISOCKET_TLS_ERROR_UNSUPPORTED_SIGALG	0xF1030008	open	Yes	The implementation does not support all the selected signature algorithms.
TEE_ISOCKET_TLS_ERROR_EV_SIG_VERIFY_FAILED	0xF1030009	open	Yes	Signature verification of received attestation evidence failed.
TEE_ISOCKET_TLS_ERROR_EV_BINDING_CHECK_FAILED	0xF103000A	open	Yes	Verification of channel bindings in received attestation evidence failed.
TEE_ISOCKET_TLS_ERROR_ALERT	0xF10301XX	open, send, recv	Yes	A fatal TLS alert was received from the server. The last byte contains the alert number defined in [RFC 8446] section 6 or [RFC 5246] section 7.2.

Name	Value	Function	Fatal	Meaning
Proprietary codes	As defined in [TEE Core]	Any	Depends	The value and meaning of other codes will be defined when an implementation is supporting TLS modes outside of the subset defined in this specification.

762

763 Proprietary error codes SHALL follow the numbering scheme described in [TEE Core] section 3.3.1, Return
764 Code Ranges and Format.

765 ### C.2.9 Instance Specific ioctl commandCode

766 **Table C-27: TLS Instance Specific ioctl commandCode**

Name	Value	Argument Type	Description
`TEE_TLS_BINDING_INFO`	`0x67000001`	`[inout] char *buf`	Retrieve channel binding information for the current connection. The returned buffer can be interpreted as an instance of the structure `TEE_tlsSocket_CB_Data`. If no channel binding information is available, the output length SHALL be set to zero. When TLS 1.3 has been negotiated for the connection, the input buffer can be used to supply the label argument for the `TLS-Exporter` mechanism. • If no label is provided, the value returned SHALL be the `tls-exporter` channel bindings defined in [RFC 9266]. • If the TA intends to use the channel bindings for post-handshake attestation, the TA SHALL NOT provide a label. If the provided buffer is too small, the implementation SHALL return `TEE_ERROR_SHORT_BUFFER`.

Name	Value	Argument Type	Description
TEE_TLS_SESSION_INFO	0x67000002	[inout] char *buf	Retrieve information about the current TLS session. The returned buffer can be interpreted as an instance of the structure TEE_tlsSocket_SessionInfo. The first octet of the input buffer SHALL be an unsigned integer indicating the desired version of the TEE_tlsSocket_SessionInfo structure to be returned. If no TLS session has been established at the time of calling (e.g. the handshake has not finished), the output length SHALL be set to zero. If the provided buffer is too small, the implementation SHALL return TEE_ERROR_SHORT_BUFFER.
TEE_TLS_RELEASE_CERT_CHAIN	0x67000003		Indicate to the implementation that it may release memory pointing to stored server certificate chain. The buf argument is ignored. Note that after this operation, it will not be possible to retrieve the server certificate chain using the TEE_TLS_SESSION_INFO command. If the storeServerCertChain option was not enabled in the session options, this command has no effect.
TEE_TLS_PEER_EVIDENCE	0x67000004	[out] char *buf	Return attestation evidence received from the remote endpoint. If no evidence was received in the handshake, the output length SHALL be set to zero. If the provided buffer is too small, the implementation SHALL return TEE_ERROR_SHORT_BUFFER.

767

C.3 Specification Properties

The properties listed in Table C-28 can be retrieved by the generic Property Access Function with the `TEE_PROPSET_TEE_IMPLEMENTATION` pseudo-handle (see [TEE Core]).

Table C-28: Specification Reserved Properties

Name	Type	Comment
`gpd.tee.tls.handshake`	`integer`	Property that indicates supported additional TLS handshake types. For values, see Table C-1.
`gpd.tee.tls.auth.remote.credential`	`integer`	Property that indicates supported credential type for remote endpoint authentication. For values, see Table C-2.
`gpd.tee.tls.auth.remote.validation_steps`	`integer`	Property that indicates supported certification path validation steps for remote server authentication. For values, see Table C-16.
`gpd.tee.tls.auth.local.credential`	`integer`	Property that indicates supported credential type for client authentication. For values, see Table C-3.
`gpd.tee.sockets.tls.version`	`integer`	Property that indicates the version number of this specification that the implementation conforms to. See section C.1.2.

The integers should have 32 bits defined and so should be retrieved via the `TEE_GetPropertyAsU32` interface.

C.4 Header File Example

```
#ifndef TEE_ISOCKET_PROTOCOLID_TLS
#include “tee_isocket.h”

// This is the current draft header file for Annex C v1.2 development.
// To see changes compared to v1.1, search for "ADDED"

/* Protocol identifier */
#define TEE_ISOCKET_PROTOCOLID_TLS 0x67

/* Instance specific errors */
#define TEE_ISOCKET_TLS_ERROR_REJECTED_SUITE            0xF1030001
#define TEE_ISOCKET_TLS_ERROR_VERSION                   0xF1030002
#define TEE_ISOCKET_TLS_ERROR_UNSUPPORTED_SUITE         0xF1030003
#define TEE_ISOCKET_TLS_ERROR_HANDSHAKE                 0xF1030004
#define TEE_ISOCKET_TLS_ERROR_AUTHENTICATION            0xF1030005
#define TEE_ISOCKET_TLS_ERROR_DATA                      0xF1030006
#define TEE_ISOCKET_TLS_ERROR_UNSUPPORTED_KEYEX_GROUP   0xF1030007
#define TEE_ISOCKET_TLS_ERROR_UNSUPPORTED_SIGALG        0xF1030008
/* ADDED in v1.2: */
#define TEE_ISOCKET_TLS_ERROR_EV_SIG_VERIFY_FAILED      0xF1030009
#define TEE_ISOCKET_TLS_ERROR_EV_BINDING_CHECK_FAILED   0xF103000A

#define TEE_ISOCKET_TLS_ERROR_ALERT(code) (0xF1030100 | ((code) & 0xFF))

/* Instance specific ioctl functions */
#define TEE_TLS_BINDING_INFO                    0x67000001
#define TEE_TLS_SESSION_INFO                    0x67000002
#define TEE_TLS_RELEASE_CERT_CHAIN              0x67000003
#define TEE_TLS_PEER_EVIDENCE                   0x67000004 /* ADDED in v1.2 */

/*
 * Structs and enums for the setup
 */

typedef uint32_t TEE_tlsSocket_TlsVersion;
#define TEE_TLS_VERSION_ALL    0x00000000
#define TEE_TLS_VERSION_1v2    0x00000001
#define TEE_TLS_VERSION_PRE1v2 0x00000002
#define TEE_TLS_VERSION_1v3    0x00000004

/* Ciphersuite list termination. */
#define TEE_TLS_NULL_WITH_NULL_NULL      0x00000000

/* TLS 1.3 ciphersuites. */
typedef uint32_t * TEE_tlsSocket_CipherSuites_GroupB;
```

```
#define TEE_TLS_AES_128_GCM_SHA256       0x00001301
#define TEE_TLS_AES_256_GCM_SHA384       0x00001302
#define TEE_TLS_CHACHA20_POLY1305_SHA256 0x00001303
#define TEE_TLS_AES_128_CCM_SHA256       0x00001304
#define TEE_TLS_AES_128_CCM_8_SHA256     0x00001305

/* Ciphersuites for TLS 1.2 and below */
typedef uint32_t *TEE_tlsSocket_CipherSuites_GroupA;
#define TEE_TLS_RSA_WITH_3DES_EDE_CBC_SHA           0x0000000A /* [RFC5246] */
#define TEE_TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA       0x00000013 /* [RFC5246] */
#define TEE_TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA       0x00000016 /* [RFC5246] */
#define TEE_TLS_RSA_WITH_AES_128_CBC_SHA            0x0000002F /* [RFC5246] */
#define TEE_TLS_DHE_DSS_WITH_AES_128_CBC_SHA        0x00000032 /* [RFC5246] */
#define TEE_TLS_DHE_RSA_WITH_AES_128_CBC_SHA        0x00000033 /* [RFC5246] */
#define TEE_TLS_RSA_WITH_AES_256_CBC_SHA            0x00000035 /* [RFC5246] */
#define TEE_TLS_DHE_DSS_WITH_AES_256_CBC_SHA        0x00000038 /* [RFC5246] */
#define TEE_TLS_DHE_RSA_WITH_AES_256_CBC_SHA        0x00000039 /* [RFC5246] */
#define TEE_TLS_RSA_WITH_AES_128_CBC_SHA256         0x0000003C /* [RFC5246] */
#define TEE_TLS_RSA_WITH_AES_256_CBC_SHA256         0x0000003D /* [RFC5246] */
#define TEE_TLS_DHE_DSS_WITH_AES_128_CBC_SHA256     0x00000040 /* [RFC5246] */
#define TEE_TLS_DHE_RSA_WITH_AES_128_CBC_SHA256     0x00000067 /* [RFC5246] */
#define TEE_TLS_DHE_DSS_WITH_AES_256_CBC_SHA256     0x0000006A /* [RFC5246] */
#define TEE_TLS_DHE_RSA_WITH_AES_256_CBC_SHA256     0x0000006B /* [RFC5246] */
#define TEE_TLS_PSK_WITH_3DES_EDE_CBC_SHA           0x0000008B /* [RFC4279] */
#define TEE_TLS_PSK_WITH_AES_128_CBC_SHA            0x0000008C /* [RFC4279] */
#define TEE_TLS_PSK_WITH_AES_256_CBC_SHA            0x0000008D /* [RFC4279] */
#define TEE_TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA       0x0000008F /* [RFC4279] */
#define TEE_TLS_DHE_PSK_WITH_AES_128_CBC_SHA        0x00000090 /* [RFC4279] */
#define TEE_TLS_DHE_PSK_WITH_AES_256_CBC_SHA        0x00000091 /* [RFC4279] */
#define TEE_TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA       0x00000093 /* [RFC4279] */
#define TEE_TLS_RSA_PSK_WITH_AES_128_CBC_SHA        0x00000094 /* [RFC4279] */
#define TEE_TLS_RSA_PSK_WITH_AES_256_CBC_SHA        0x00000095 /* [RFC4279] */
#define TEE_TLS_RSA_WITH_AES_128_GCM_SHA256         0x0000009C /* [RFC5288] */
#define TEE_TLS_RSA_WITH_AES_256_GCM_SHA384         0x0000009D /* [RFC5288] */
#define TEE_TLS_DHE_RSA_WITH_AES_128_GCM_SHA256     0x0000009E /* [RFC5288] */
#define TEE_TLS_DHE_RSA_WITH_AES_256_GCM_SHA384     0x0000009F /* [RFC5288] */
#define TEE_TLS_DHE_DSS_WITH_AES_128_GCM_SHA256     0x000000A2 /* [RFC5288] */
#define TEE_TLS_DHE_DSS_WITH_AES_256_GCM_SHA384     0x000000A3 /* [RFC5288] */
#define TEE_TLS_PSK_WITH_AES_128_GCM_SHA256         0x000000A8 /* [RFC5487] */
#define TEE_TLS_PSK_WITH_AES_256_GCM_SHA384         0x000000A9 /* [RFC5487] */
#define TEE_TLS_DHE_PSK_WITH_AES_128_GCM_SHA256     0x000000AA /* [RFC5487] */
#define TEE_TLS_DHE_PSK_WITH_AES_256_GCM_SHA384     0x000000AB /* [RFC5487] */
#define TEE_TLS_RSA_PSK_WITH_AES_128_GCM_SHA256     0x000000AC /* [RFC5487] */
#define TEE_TLS_RSA_PSK_WITH_AES_256_GCM_SHA384     0x000000AD /* [RFC5487] */
#define TEE_TLS_PSK_WITH_AES_128_CBC_SHA256         0x000000AE /* [RFC5487] */
#define TEE_TLS_PSK_WITH_AES_256_CBC_SHA384         0x000000AF /* [RFC5487] */
#define TEE_TLS_DHE_PSK_WITH_AES_128_CBC_SHA256     0x000000B2 /* [RFC5487] */
```

```
#define TEE_TLS_DHE_PSK_WITH_AES_256_CBC_SHA384      0x000000B3 /* [RFC5487] */
#define TEE_TLS_RSA_PSK_WITH_AES_128_CBC_SHA256      0x000000B6 /* [RFC5487] */
#define TEE_TLS_RSA_PSK_WITH_AES_256_CBC_SHA384      0x000000B7 /* [RFC5487] */
#define TEE_TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA    0x0000C008 /* [RFC4492] */
#define TEE_TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA     0x0000C009 /* [RFC4492] */
#define TEE_TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA     0x0000C00A /* [RFC4492] */
#define TEE_TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA      0x0000C012 /* [RFC4492] */
#define TEE_TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA       0x0000C013 /* [RFC4492] */
#define TEE_TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA       0x0000C014 /* [RFC4492] */
#define TEE_TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA        0x0000C01A /* [RFC5054] */
#define TEE_TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA    0x0000C01B /* [RFC5054] */
#define TEE_TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA    0x0000C01C /* [RFC5054] */
#define TEE_TLS_SRP_SHA_WITH_AES_128_CBC_SHA         0x0000C01D /* [RFC5054] */
#define TEE_TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA     0x0000C01E /* [RFC5054] */
#define TEE_TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA     0x0000C01F /* [RFC5054] */
#define TEE_TLS_SRP_SHA_WITH_AES_256_CBC_SHA         0x0000C020 /* [RFC5054] */
#define TEE_TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA     0x0000C021 /* [RFC5054] */
#define TEE_TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA     0x0000C022 /* [RFC5054] */
#define TEE_TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256  0x0000C023 /* [RFC5289] */
#define TEE_TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384  0x0000C024 /* [RFC5289] */
#define TEE_TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256    0x0000C027 /* [RFC5289] */
#define TEE_TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384    0x0000C028 /* [RFC5289] */
#define TEE_TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256  0x0000C02B /* [RFC5289] */
#define TEE_TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384  0x0000C02C /* [RFC5289] */
#define TEE_TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256    0x0000C02F /* [RFC5289] */
#define TEE_TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384    0x0000C030 /* [RFC5289] */
#define TEE_TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA      0x0000C034 /* [RFC5489] */
#define TEE_TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA       0x0000C035 /* [RFC5489] */
#define TEE_TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA       0x0000C036 /* [RFC5489] */
#define TEE_TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256    0x0000C037 /* [RFC5489] */
#define TEE_TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384    0x0000C038 /* [RFC5489] */
#define TEE_TLS_RSA_WITH_AES_128_CCM                 0x0000C09C /* [RFC6655] */
#define TEE_TLS_RSA_WITH_AES_256_CCM                 0x0000C09D /* [RFC6655] */
#define TEE_TLS_DHE_RSA_WITH_AES_128_CCM             0x0000C09E /* [RFC6655] */
#define TEE_TLS_DHE_RSA_WITH_AES_256_CCM             0x0000C09F /* [RFC6655] */
#define TEE_TLS_PSK_WITH_AES_128_CCM                 0x0000C0A4 /* [RFC6655] */
#define TEE_TLS_PSK_WITH_AES_256_CCM                 0x0000C0A5 /* [RFC6655] */
#define TEE_TLS_DHE_PSK_WITH_AES_128_CCM             0x0000C0A6 /* [RFC6655] */
#define TEE_TLS_DHE_PSK_WITH_AES_256_CCM             0x0000C0A7 /* [RFC6655] */

/* Signature algorithms. */
typedef uint32_t TEE_tlsSocket_SignatureScheme;
#define TEE_TLS_RSA_PKCS1_SHA256                0x00000401
#define TEE_TLS_RSA_PKCS1_SHA384                0x00000501
#define TEE_TLS_RSA_PKCS1_SHA512                0x00000601
#define TEE_TLS_ECDSA_SECP256R1_SHA256          0x00000403
#define TEE_TLS_ECDSA_SECP384R1_SHA384          0x00000503
```

```
#define TEE_TLS_ECDSA_SECP521R1_SHA512          0x00000603
#define TEE_TLS_RSA_PSS_RSAE_SHA256             0x00000804
#define TEE_TLS_RSA_PSS_RSAE_SHA384             0x00000805
#define TEE_TLS_RSA_PSS_RSAE_SHA512             0x00000806
#define TEE_TLS_ED25519                         0x00000807
#define TEE_TLS_ED448                           0x00000808
#define TEE_TLS_RSA_PSS_PSS_SHA256              0x00000809
#define TEE_TLS_RSA_PSS_PSS_SHA384              0x0000080A
#define TEE_TLS_RSA_PSS_PSS_SHA512              0x0000080B
#define TEE_TLS_RSA_PKCS_SHA1                   0x00000201
#define TEE_TLS_ECDSA_SHA1                      0x00000203

/* Key exchange groups used in TLS 1.3 */
typedef uint32_t TEE_tlsSocket_Tls13KeyExGroup;
#define TEE_TLS_KEYEX_GROUP_SECP256R1   0x00000017
#define TEE_TLS_KEYEX_GROUP_SECP384R1   0x00000018
#define TEE_TLS_KEYEX_GROUP_SECP521R1   0x00000019
#define TEE_TLS_KEYEX_GROUP_X25519      0x0000001D
#define TEE_TLS_KEYEX_GROUP_X4458       0x0000001E
#define TEE_TLS_KEYEX_GROUP_FFDHE_2048  0x00000100
#define TEE_TLS_KEYEX_GROUP_FFDHE_3072  0x00000101
#define TEE_TLS_KEYEX_GROUP_FFDHE_4096  0x00000102
#define TEE_TLS_KEYEX_GROUP_FFDHE_6144  0x00000103
#define TEE_TLS_KEYEX_GROUP_FFDHE_8192  0x00000104

/* The definition below is just a simple example of what an implementation
   could define. */
typedef struct TEE_tlsSocket_Context_s {
    /*
     * All things needed to maintain the context
     */
    uint32_t protocolError;
    uint32_t state;
} TEE_tlsSocket_Context;

typedef struct TEE_tlsSocket_PSK_Info_s {
    TEE_ObjectHandle    pskKey;
    char                *pskIdentity;
} TEE_tlsSocket_PSK_Info;


typedef struct TEE_tlsSocket_SRP_Info_s {
    char *srpPassword;
    char *srpIdentity;
} TEE_tlsSocket_SRP_Info;

typedef struct TEE_tlsSocket_ClientPDC_s {
```

```
    TEE_ObjectHandle    privateKey;
    uint8_t             *bulkCertChain;
    uint32_t            bulkSize;
    uint32_t            bulkEncoding;
} TEE_tlsSocket_ClientPDC;


typedef struct TEE_tlsSocket_ServerPDC_s {
    TEE_ObjectHandle    publicKey;
    // The following fields were introduced in v1.1
    TEE_ObjectHandle    *trustedCerts;
    uint32_t            *trustedCertEncodings;
    uint32_t            numTrustedCerts;
    uint32_t            allowTAPersistentTimeCheck;
    uint8_t             *certPins;
    uint32_t            numCertPins;
    uint8_t             *pubkeyPins;
    uint32_t            numPubkeyPins;
} TEE_tlsSocket_ServerPDC;

typedef uint32_t TEE_tlsSocket_ClientCredentialType;
#define TEE_TLS_CLIENT_CRED_NONE 0x00000000
#define TEE_TLS_CLIENT_CRED_PDC  0x00000001
#define TEE_TLS_CLIENT_CRED_CSC  0x00000002

typedef uint32_t TEE_tlsSocket_ServerCredentialType;
#define TEE_TLS_SERVER_CRED_PDC         0x00000000
#define TEE_TLS_SERVER_CRED_CSC         0x00000001
#define TEE_TLS_SERVER_CRED_CERT_PIN    0x00000002
#define TEE_TLS_SERVER_CRED_PUBKEY_PIN  0x00000003

typedef struct TEE_tlsSocket_Credentials_s {
    TEE_tlsSocket_ServerCredentialType  serverCredType;
    TEE_tlsSocket_ServerPDC             *serverCred;
    TEE_tlsSocket_ClientCredentialType  clientCredType;
    TEE_tlsSocket_ClientPDC             *clientCred;
} TEE_tlsSocket_Credentials;

/* ADDED in v1.2: */
typedef uint32_t TEE_tlsSocket_AttEvTransMethod;
# define TEE_TLS_ATT_EV_TRANS_METHOD_NONE               0x00000000
# define TEE_TLS_ATT_EV_TRANS_METHOD_X509_EXTENSION     0x00000001
# define TEE_TLS_ATT_EV_TRANS_METHOD_EXTRA_CERT         0x00000002
# define TEE_TLS_ATT_EV_TRANS_METHOD_CERT_MSG_EXT       0x00000004

/* ADDED in v1.2: */
typedef uint32_t TEE_tlsSocket_AttFlags;
```

```
# define TEE_TLS_ATT_FLAG_SEND_EVIDENCE             0x00000001
# define TEE_TLS_ATT_FLAG_SEND_UNSOLICITED_EVIDENCE 0x00000002
# define TEE_TLS_ATT_FLAG_REQUEST_EVIDENCE          0x00000004
# define TEE_TLS_ATT_FLAG_REQUIRE_EVIDENCE          0x00000008
# define TEE_TLS_ATT_FLAG_PRIVACY                   0x00000010
# define TEE_TLS_ATT_FLAG_USE_ATTESTATION_SERVICE   0x00000020

/* ADDED in v1.2: */
typedef struct TEE_tlsSocket_AttestationSetup_s {
  TEE_tlsSocket_AttFlags          flags;
  TEE_tlsSocket_AttEvTransMethod  sendEvTransMethod;
  TEE_tlsSocket_AttEvTransMethod  recvEvTransMethod;
  TEE_tlsSocket_Credentials       *evidenceCred;
  TEE_UUID                        *attEnvUUID;
} TEE_tlsSocket_AttestationSetup;

/*
 * Struct for retrieving channel binding data
 * using the ioctl functionality.
 */
typedef struct TEE_tlsSocket_CB_Data_s {
    uint32_t  cb_data_size;
    uint8_t   cb_data[];
} TEE_tlsSocket_CB_Data;

/*
 * Struct for retrieving session information
 * using the ioctl functionality.
 */

typedef struct TEE_tlsSocket_SessionInfo_s
{
    uint8_t                         structVersion;
    TEE_tlsSocket_TlsVersion        chosenVersion;
    uint32_t                        chosenCiphersuite;
    TEE_tlsSocket_SignatureScheme   chosenSigAlg;
    TEE_tlsSocket_Tls13KeyExGroup   chosenKeyExGroup;
    unsigned char                   *matchedServerName;
    uint32_t                        matchedServerNameLen;
    const uint8_t                   *validatedServerCertificate;
    uint32_t                        validatedServerCertificateLen;
    uint32_t                        usedServerAuthenticationMethod;
    // The following fields were introduced in v1.2:
    TEE_tlsSocket_AttEvTransMethod  recvAttestationType;
} TEE_tlsSocket_SessionInfo;

/* Structure for storing session tickets. */
```

```
typedef struct TEE_tlsSocket_SessionTicket_Info_s {
    uint8_t                  *encrypted_ticket;
    uint32_t                 encrypted_ticket_len;
    uint8_t                  *server_id;
    uint32_t                 server_id_len;
    uint8_t                  *session_params;
    uint32_t                 session_params_len;
    uint8_t                  caller_allocated;
    TEE_tlsSocket_PSK_Info   psk;
} TEE_tlsSocket_SessionTicket_Info;

/* The TEE TLS setup struct */
typedef struct TEE_tlsSocket_Setup_s {
    TEE_tlsSocket_TlsVersion acceptServerVersion;
    TEE_tlsSocket_CipherSuites_GroupA *allowedCipherSuitesGroupA;
    TEE_tlsSocket_PSK_Info *PSKInfo;
    TEE_tlsSocket_SRP_Info *SRPInfo;
    TEE_tlsSocket_Credentials *credentials;
    TEE_iSocket *baseSocket;
    TEE_iSocketHandle *baseContext;

    // The following fields were introduced in v1.1
    TEE_tlsSocket_CipherSuites_GroupB *allowedCipherSuitesGroupB;
    TEE_tlsSocket_SignatureScheme *sigAlgs;
    uint32_t numSigAlgs;
    TEE_tlsSocket_SignatureScheme *certSigAlgs;
    uint32_t numCertSigAlgs;
    TEE_tlsSocket_Tls13KeyExGroup *tls13KeyExGroups;
    uint32_t numTls13KeyExGroups;
    uint32_t numTls13KeyShares;
    TEE_tlsSocket_SessionTicket_Info  *sessionTickets;
    uint32_t sessionTicketsNumElements;
    uint32_t numStoredSessionTickets;
    unsigned char *serverName;
    uint32_t serverNameLen;
    uint8_t *serverCertChainBuf;
    uint32_t *serverCertChainBufLen;
    uint8_t storeServerCertChain;
    unsigned char **alpnProtocolIds;
    uint32_t *alpnProtocolIdLens;
    uint32_t numAlpnProtocolIds;

     // The following fields were introduced in v1.2
    TEE_tlsSocket_AttestationSetup *attestationSetup; /* ADDED in v1.2 */

} TEE_tlsSocket_Setup;
```

```
/* declare the function pointer handle */
extern TEE_iSocket * const TEE_tlsSocket;
#endif
```

C.5 Additional Cipher Suite References

A TLS cipher suite constant defines three entities:

- The authentication and key exchange algorithm
- The bulk encryption algorithm (cipher and mode)
- The message authentication algorithm

The tables below list the supported algorithms for each entity.

See section C.2.6.2 for a detailed description of the constants.

Note: This version of the specification only supports ephemeral Diffie-Hellman, as the TEE currently has no way of interpreting certificates. This may change in future versions of specifications.

Table C-29: Supported Authentication and Key Exchange Algorithms

Algorithm	Main Reference
Pre-shared key (PSK)	[RFC 4279]
PSK with ephemeral Diffie-Hellman	
PSK with server side RSA certificate	
Secure remote password (SRP)	[RFC 5054]
SRP with server side RSA certificate	
SRP with server side DSS certificate	
Server side RSA certificate	[RFC 5246]
Ephemeral Diffie-Hellman with server side RSA certificate	
Ephemeral Diffie-Hellman with server side DSS certificate.	
PSK with Ephemeral Elliptic Curve Diffie-Hellman	[RFC 5489]
Ephemeral Elliptic Curve Diffie-Hellman with server side RSA certificate	[RFC 5289]
Ephemeral Elliptic Curve Diffie-Hellman with server side ECDSA certificate	[RFC 4492]

Table C-30: Supported Bulk Encryption Algorithms

Algorithm	Main Reference
Triple-DES with 112-bit key in CBC mode	[RFC 5246]
AES with 128-bit key in CBC mode	
AES with 256-bit key in CBC mode	
AES with 128-bit key in CCM mode providing both confidentiality and authenticity	[RFC 6655]
AES with 256-bit key in CCM mode providing both confidentiality and authenticity	
AES with 128-bit key in GCM mode providing both confidentiality and authenticity	[RFC 5288]
AES with 256-bit key in GCM mode providing both confidentiality and authenticity	

1121 **Table C-31: Supported Message Authentication Algorithms**

Algorithm	Main Reference
CCM or GCM. This bulk encryption mode provides both encryption and message authentication.	[RFC 6655], [RFC 5288]
HMAC with SHA-1	[RFC 5246]
HMAC with SHA-256	
HMAC with SHA-384	

1122