Apple Trust Policy Module Functional Specification

2005-01-25

Apple Trust Policy Module Functional Specification

January 25, 2005

1.0 Scope

This document describes the functions of the Apple Trust Policy Module ("the TP") for the Mac OS X platform.

2.0 Functional Specification

The TP's functional description mainly describes the behavior of the CSSM_TP_CertGroupVerify function, which is currently the primary (possibly the only) function currently used by other Apple software. The function's API is defined here and the mechanics of the trust policies it implements are described.

The CSSM_TP_CertGroupConstruct function is also supported; its use can easily be inferred from the following discussion, as it is merely a subset of the CertGroupVerify function (without the policy enforcement).

2.1 The CSSM_TP_CertGroupVerify function

This function basically performs two tasks. First it takes an unordered group of certificates ("certs") and an optional list of open DB handles, and attempts to create a valid, ordered cert chain starting from the first cert passed in and terminating at a either a root cert or a cert which is verifiable by one of a set of optional Anchor Certs specified by the caller. The list of DBs passed in will be searched for any certs needed to complete the chain. Certiicates may also optionally be fetched from the net while creating the ordered list of certificates if so enabled (see section 2.1, "CSSM_TP_CertGroupVerify Calling syntax"). The resulting ordered cert chain is optionally passed back to the caller. The description of an "ordered cert chain" can be found in section 2.3.

Second, this function performs enforcement of one of a number of Trust Policies ("Policies"). Details of these policies are found in section 2.4. The enforcement of the policy is performed on the ordered cert chain constructed as described above.

2.1 CSSM_TP_CertGroupVerify Calling syntax

```
CSSM_RETURN CSSM_TP_CertGroupVerify (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CSP_HANDLE CSPHandle,
        const CSSM_CERTGROUP *CertGroupToBeVerified,
        const CSSM_TP_VERIFY_CONTEXT *VerifyContext,
        CSSM_TP_VERIFY_CONTEXT_RESULT_PTR VerifyContextResult);
```

Arguments are as follows:

Three handles - to an open TP, CL, and CSP. The CSP must be capable of dealing with the signature algorithms used by the certs. The CL must be an X.509-savvy CL. It's up to the calling app to sort this out. Currently the standard AppleCSP and AppleX509CL will suffice.

CertGroupToBeVerified contains an unordered array of raw certs in the form of a CSSM_CERTGROUP_PTR. The first cert of this list is the subject cert (leaf or end entity) which is eventually to be verified. The other certs can be in any order and may not even have any relevance to the cert chain being constructed. They may also be invalid certs. (If the first cert is invalid, the function will return CSSM_TP_INVALID_CERTIFICATE.) The contents of CertGroupToBeVerified must be:

CertType	must be CSSM_CERT_X_509{v1,v2,v3}
CertEncoding	must be CSSM_CERT_ENCODING{DER,BER}
NumCerts	is the number of certs
CertList	is an array of CSSM_DATAs, each containing one DER or BER encoded cert
CertGroupType	must be CSSM_CERTGROUP_DATA

VerifyContext contains several fields, starting with CSSM_TP_CALLERAUTH_CONTEXT_PTR **Cred**, which in turn has the following interesting fields:

CSSM_TP_POLICY_INFO **Policy**

Policy.NumberOfPolicyIds specifies the number of policies and must be zero or one.

Policy.PolicyIds contains an optional array of Policy OID fields (if present, the size of the array must be one). The Policy OIDs are defined in <Security/oidsalg.h> and are discussed below in section 2.4, "Trust Policies". The FieldValue field in the PolicyField is optional, on a per-policy basis; the values are described below.

Policy.PolicyControl is currently unused.

NumberOfAnchorCerts and **AnchorCerts** are an optional array of "known trusted certs" which may anchor a cert chain. If no anchors are presented here, there is no way this function can return CSSM_OK.

DBList must be present (this field is a pointer). It must contain {0,NULL} if no DL/DB list is to be searched. (This is required because the TP_CertGroupConstruct function interface, a public function which is called by the implementation of TP_CertGroupVerify, contains this as a required field.) If a valid DLDB list is specified, that list is searched for certs to complete the cert chain.

CallerCredentials must be NULL.

VerifyTime is an optional pointer to a CSSM-formatted time string indicating the time at which the cert chain is to be evaluated. If the pointer is NULL, the cert chain is evaluated "now".

All other CSSM_TP_CALLERAUTH_CONTEXT fields are ignored.

VerifyContext->ActionData.Data contains an optional pointer to a CSSM_APPLE_TP_ACTION_DATA struct. Currently the only field defined for this struct is a word of flags called CSSM_APPLE_TP_ACTION_FLAGS, defined in <Security/cssmapple.h>. The bits in this word specify options which apply to all policies' they are defined as follows:

CSSM_TP_ACTION_ALLOW_EXPIRED: when set, indicates that certs' "not after" fields are to be ignored.

CSSM_TP_ACTION_LEAF_IS_CA: when set, indicates that the leaf cert is known to be a CA. Normally this situation results in a policy verification error if the leafd cert contains a BasicConstraints extension with the CA bit set. This flag allows verification of such a cert as leaf.

CSSM_TP_ACTION_FETCH_CERT_FROM_NET: when set, enabled the TP to attempt to find issuer certs from the net. When so enabled, this occurs if a cert appears in a cert chain, and no issuer is found locally (in the incoming certs or in the optional DBList), and the cert has an IssuerAltName extension with a URI component.

CSSM_TP_ACTION_ALLOW_EXPIRED_ROOT: when set, indicates that root certs' "not after" fields are to be ignored.

CSSM_TP_ACTION_REQUIRE_REV_PER_CERT: when set, then some positive revocation check must be successfully be performed for each certificate except for roots. See section 2.4.2 for information on revocation policies.

VerifyContextResult, if non-NULL, on return contains the result of the operation in the form of an ordered cert chain along with detailed per-certificate status information allowing the caller to determine exactly where in the cert chain errors might have occurred. This info is returned regardless of the outcome of policy verification. If caller specifies a NULL VerifyContextResult pointer, no evidence will be returned The contents are as follows:

NumberOfEvidences = 3

Evidence[0] . EvidenceForm**:** CSSM_EVIDENCE_FORM_APPLE_HEADER
Evidence[0] . Evidence: ptr to CSSM_TP_APPLE_EVIDENCE_HEADER

> This header currently contains one field, Version, with a value of CSSM_TP_APPLE_EVIDENCE_VERSION (0). This struct and all of its contents must be freed by the caller using the standard app-level memory callback.

Evidence[1] . EvidenceForm**:** `CSSM_EVIDENCE_FORM_APPLE_CERTGROUP`
Evidence[1] . Evidence: ptr to `CSSM_CERTGROUP`

> This field contains the resulting cert group in the same format as the CSSM_CERTGROUP passed in as **CertGroupToBeVerified.** This struct and all of its contents must be freed by the caller using the standard app-level memory callback.

Evidence[2] . EvidenceForm**:** `CSSM_EVIDENCE_FORM_APPLE_CERT_INFO`
Evidence[1] . Evidence: ptr to `CSSM_TP_APPLE_EVIDENCE_INFO`

> The evidence pointer refers to an array of `CSSM_TP_APPLE_EVIDENCE_INFO`s which is the same size (in elements) as the certs in Evidence[1].Evidence. There is a one-to-one correspondence between the elements of the `CSSM_TP_APPLE_EVIDENCE_INFO` and the certs in the cert group in Evidence[1].Evidence. See <Security/cssmapple.h> for a description of the contents of this struct. It provides for an arbitrary number of CSSM_RETURN codes to be assigned to a given cert via the StatusCodes field. The StatusCodes field, as well as the `CSSM_TP_APPLE_EVIDENCE_INFO` array itself, must be freed by the caller using the standard app-level memory callback.

2.2 CSSM_TP_CertGroupVerify Return Values

CSSM_OK : Cert chain verified all the way back to an Anchor cert and (optional) policy enforcement succeeded. Note that in this case, depending on the requested policy, a root cert may or may not have been in the incoming CertGroup or in one of the open Dbs. The root cert was definitely in the list of AnchorCerts.

CSSMERR_TP_INVALID_ANCHOR_CERT: In this case, the cert chain was validated back to a self-signed (root) cert found in either incoming cert group or in one of the DBs in DBList, but that root cert was **not** found in the AnchorCert list.

CSSMERR_TP_NOT_TRUSTED: No root cert found, and the last cert in the chain was not verifiable by any certs in AnchorCerts.

CSSMERR_TP_VERIFICATION_FAILURE: A root cert was found which did not self-verify.

CSSMERR_TP_VERIFY_ACTION_FAILED: The requested policy verification failed

CSSMERR_TP_INVALID_CERTIFICATE: Bad leaf certificate.

CSSMERR_TP_CERT_EXPIRED and **CSSMERR_TP_CERT_NOT_VALID_YET**: One of the certificates in the chain has expired or is not yet valid.

CSSMERR_TP_INVALID_REQUEST_INPUTS : no incoming VerifyContext.

Other policy-specific return values may also be returned; see section 2.4.

2.3 Ordered Cert Chain Construction

When attempting to construct an ordered cert chain, the first thing the TP does is to match a cert's issuer name to another cert's subject name. The order of the constructed cert chain is such that cert[0] is the end entity, or leaf, cert; thus; for each cert *n* in the chain, the issuer of cert[*n*] is the same as the subject of cert[*n*+1]. The exception to this is when a root cert is found. A root cert has identical values for subject and issuer. Subject and Issuer names are normalized, per RFC 3280, section 4.1.2.4, prior to the construction of an ordered cert chain. (Normalizing involves munging of printable strings to upper case and removing leading, trailing, and redundant whitespace.) Issuer certs can come from the incoming cert group, or a DLDBList, or from the network (the latter two are optional).

Also, for each cert n in the chain, the public key of cert[*n*+1] must successfully perform a signature verification of the tbsCertificate portion of cert[*n*]. Again, a root cert is different; a root cert must successfully perform signature verification on itself. Signature verification is used to resolve ambiguities in the subject/issuer chain, as for example when more than one cert with a matching subject name is found for a given cert's issuer.

Additionally, the "not before" and "not after" fields of each cert in the chain must be valid. The CSSM_TP_ALLOW_EXPIRE actionData flag can be used to ignore the (unfortunately) common case of expired certs. When this option is specified, the "not after" field of all certs is ignored.

2.4 Trust Policies

Trust policy verification involves parsing and evaluating the extensions in the cert chain. The default policy does not do this.

The policy OIDs, defined in <Security/oidsalg.h>, are as follows:

AppleBasic:	CSSMOID_APPLE_X509_BASIC
iSign:	CSSMOID_APPLE_ISIGN
SSL:	CSSMOID_APPLE_TP_SSL
S/MIME:	CSSMOID_APPLE_TP_SMIME
EAP:	CSSMOID_APPLE_TP_EAP
CodeSigning:	CSSMOID_APPLE_TP_CODE_SIGN
IPsec:	CSSMOID_APPLE_TP_IP_SEC
IChat:	CSSMOID_APPLE_TP_ICHAT
CRL:	CSSMOID_APPLE_TP_REVOCATION_CRL
OCSP:	CSSMOID_APPLE_TP_REVOCATION_OCSP

Refer to RFC 3280 for descriptions of the various cert extensions discussed below.

Some policies allow for additional parameters to be specified in the VerifyContext->Cred->Policy.PolicyIds->FieldValue.Data field. This optionally points to a policy-specific struct (see policy descriptions, below).

See section 2.4.2 for discussion of the CRL and OCSP policies.

2.4.1 Certificate Extensions

The requirements for some common extensions extensions are as follows. In all cases, a failure of the policy enforcement results in a CSSMERR_TP_VERIFY_ACTION_FAILED error as well as error-specific information being associated with the errant cert in the (optionally) returned `CSSM_TP_APPLE_EVIDENCE_INFO` array.

Critical flag

All policies: If an extension is found which the TP does not understand and which is flagged "critical", the policy check fails.

Basic Constraints

Required in all but root and leaf cert. Optional in root, with a default cA of true if not present. Optional in leaf with a default cA of false.

cA (either explicitly or by default if not present) must be false for leaf certs, true for all others. App can override the failure resulting from this bit being set for the leaf cert by asserting CSSM_TP_ACTION_LEAF_IS_CA.

PathLengthConstraint, if present (it's optional) is always enforced per RFC 3280.

Note: RFC 3280 says that CA certs MUST contain BasicConstraints extensions and that the extension MUST be flagged critical. In the real world this is not feasible. Verisign has intermediate certs with a BasicConstraints flagged not critical, and some RSA root certs (which are CA by default) do not contain this extension at all. Thus the critical flag in BasicConstraints is ignored by the TP and root certs do not have to have a BasicConstraints extension at all.

Authority Key Identifier

iSign: Optional, but illegal for root. If present, must not be critical.
All: see below for Key Identifier chain validation.

Subject Key Identifier

iSign: If present, must not be critical.
All: see below for Key Identifier chain validation.

Key Usage

iSign: Required for all except root and leaf. Key Usage for leaf must have digitalSignature bit set. Exception: a leaf cert without a Key Usage extension is legal if a netscape-cert-type extension is present with the Object Signing bit set.
CodeSigning: optional; if present in leaf, usage must have digitalSignature bit set.
All others: optional. Usage bits ignored for leaf cert. (**Note**: this does not guarantee that the leaf cert can be used for any intended purpose, as the CL will generate rather rigorous KeyUsage values based on this field and the Extended Key Usage field when extracting a key, and the CSP will enforce those usage bits.)
All: If present in non-leaf, must have keyCertSign bit set.

Extended key usage

iSign: if present in leaf, must have one usage, CSSMOID_ExtendedUseCodeSigning.
Otherwise: ignored at this point; may be checked on per-policy basis (see below).

Authority Key Identifier → Subject Key Identifier linkage

For all policies, if these (optional) fields are present, they are verified in sequence in a manner similar to the way in which the subject/issuer chain is verified. The keyIdentifier component of the Authority Key Identifier is used in chain verification. In this case, though, if either field is absent (i.e., Authority Key ID for cert[n] or Subject Key ID for cert[n+1] is missing, the verification doesn't fail; it's skipped for that step. The raw value of the encoded keyIdentifier from the Authority Key ID is compared to the raw encoded Subject Key ID.

2.4.1 Policy-specific checks

AppleBasic
iSign

No further policy-specific processing.

SSL
EAP
IPsec

These three policies currently behave identically. They take an optional pointer to a policy-specific struct, CSSM_APPLE_TP_SSL_OPTIONS, in VerifyContext->Cred->Policy.PolicyIds->FieldValue.Data. The fields in this struct are:

Version: must be CSSM_APPLE_TP_SSL_OPTS_VERSION

ServerName, **ServerNameLen**: optionally specifies the domain name of the server associated with the leaf cert. See below.

Flags: currently only one bit is defined, CSSM_APPLE_TP_SSL_CLIENT which when set tells the TP that the leaf cert belongs to a client, not a server. This is used when evaluating Extended Key Use extensions (see below).

If the ServerName and ServerNameLen fields are present (the TP acts as if they are not present if the entire CSSM_APPLE_TP_SSL_OPTIONS is absent), they express the domain name of a host, in ASCII, like "store.apple.com". Various components in the leaf certificate are compared against this string to enforce the match of "expected host name" (passed in by the app) and "found host name" (in the cert). Refer to RFC 2818, section 3.1 for detailed info on this comparison. The TP implements this section of the RFC to the letter, including proper handling of IP-address formatted host name ("17.254.3.41") and wildcards in commonName fields ("*.apple.com"). A failure of this hostname checking results in a return value of CSSMERR_APPLETP_HOSTNAME_MISMATCH from CSSM_TP_CertGroupVerify().

The TP then examines the leaf cert for an Extended Key Use extension. If there is an EKU extension, then at least one appropriate EKU OID value must be present: either an CSSMOID_ExtendedKeyUsageAny, or else CSSMOID_ServerAuth (if the leaf cert belongs to a server) or CSSMOID_ClientAuth (if the leaf cert belongs to a client). A failure of this EKU checking results in a return value of CSSMERR_TP_VERIFY_ACTION_FAILED from CSSM_TP_CertGroupVerify(). The leaf cert is flagged with a CSSMERR_APPLETP_SSL_BAD_EXT_KEY_USE status.

S/MIME

This takes an optional pointer to a policy-specific struct, CSSM_APPLE_TP_SMIME_OPTIONS, in VerifyContext->Cred->Policy.PolicyIds->FieldValue.Data. The fields in this struct are:

Version: must CSSM_APPLE_TP_SMIME_OPTS_VERSION

SenderEmail, **SenderEmailLen**: optionally specifies the email addresss of associated with the leaf cert. The TP uses this to enforce section 3 of RFC 2632 to the letter.

IntendedKeyUsage: used to express what the cert (actually, the associated key) is to be used for. The TP uses this to enforce section 4.2.1.3 of RFC 3280 when evaluating a KeyUsage extension.

NOTE: Although RFC 2632 says that a KeyUsage extension (in an S/MIME context) MUST be marked critical, this is not enforced by the TP due to the prevalence of nonconforming certificates.

Subsequent to emailAddress processing, the TP then examines the leaf cert for an Extended Key Use extension. If there is an EKU extension, then at least one appropriate EKU OID value must be present: either a CSSMOID_ExtendedKeyUsageAny, or CSSMOID_EmailProtection. A failure of this EKU checking results in a return value of CSSMERR_APPLETP_SMIME_BAD_EXT_KEY_USE from CSSM_TP_CertGroupVerify(). The leaf cert being flagged with the same status.

iChat

This cert policy is intended to be used with two disparate types of certificates. One is the .mac cert vended by Apple, and which is currently intended to be used only for iChat. Depending on the extensions in the cert, these certs can be used for iChat signing, iChat encyrption, or both. The contents of these certs has been defined elsewhere in multiple places. Refer to whatever is the currently accepted documentation for these certs.

The other type of cert which is verifiable by this policy is the generic cert which is usable in Windows AIM clients. Lacking any published spec for these certs, it has been determined that these certs contain two crucial fields:

- an EmailProtection Extended Key Usage OID, and
- a netscape-cert-type extension with the SMIME bit (0x2000) set

This policy takes an optional pointer to policy-specific struct, CSSM_APPLE_TP_SMIME_OPTIONS, in VerifyContext->Cred->Policy.PolicyIds->FieldValue.Data. The fields in this struct are:

Version: must CSSM_APPLE_TP_SMIME_OPTS_VERSION

SenderEmail, **SenderEmailLen**: optionally specifies the iChat “handle” of associated with the leaf cert. See below.

IntendedKeyUsage: optionally used to specify a required ExtendedKeyUse OID. See below.

If the application has not specified an iChat handle in the incoming

CSSM_APPLE_TP_SMIME_OPTIONS, skip this section and proceed to the section below entitled ExtendedKeyUse verification. In this case the cert is expected to conform to the generic AIM format, not the Apple-specific iChat format.

If the application has specified an iChat handle in the incoming CSSM_APPLE_TP_SMIME_OPTIONS, then the leaf certificate is examined for the presence fields which match the handle specified by the app. This is non trivial.

First the Apple iChat format is checked, in which the commonName component of SubjectName contains the .mac user name (e.g., “dmitch”) and the OrganizationUnit component contains the domain in the app-specified handle (e.g. “mac.com” but it can be anything). The incoming handle has the whole thing – “dmitch@mac.com” – and the TP breaks this into components and does the comparisons in a case insensitive manner. (Note this differs from the name matching for email addresses in S/MIME certs, in which the domain (“mac.com”) is case insensitive, but the name (“dmitch) is case **sensitive**.) The handle passed in by the app is in UTF8 format. The components in the cert can be in any encoding (T61String, BMPString, IA5String, UTF8String, etc.). In addition, the OrganizationName component of the SubjectName must contain the string “Apple Computer, Inc.”. The entire SubjectName is searched until all three components (name, component, “Apple Computer, Inc.”) are found. If they are found then henceforth this cert is presumed to be an Apple-issued iChat cert.

If the search described in the previous paragraph fails then a literal match for the incoming handle is sought, first in the SubjectAltName extension (name type RFC822Name), then in the SubjectName (component EmailAddress). All string comparisons are case-insensitive for both name and domain portions if the handle. If a match is found then this portion of the evaluation succeeds and the cert is presumed to be a generic AIM cert (not an Apple iChat cert). Proceed to ExtendedKeyUse evaluation.

If, at the end of the search described in the previous two paragraphs, no matching email addresses are found, one of two error codes is returned:

- If **at least one** email address was found in the cert, but no email address was found which matches the email address provided by the app, CSSMERR_APPLETP_SMIME_EMAIL_ADDRS_NOT_FOUND is returned.
- If **no** email addresses are found in the cert, CSSMERR_APPLETP_SMIME_NO_EMAIL_ADDRS is returned.

Note that if the app did not specify an email address in the incoming CSSM_APPLE_TP_SMIME_OPTIONS, neither of these error codes will be returned; email address checking is skipped.

ExtendedKeyUse verification in the iChat policy

Evaluation of the ExtendedKeyUse (EKU) extension depends on the type of cert – generic AIM, or Apple iChat (see above description on verification of iChat handle). An EKU extension contains a list of allowable uses, each of which is expressed as an OID.

Generic AIM certs must contain either an EmailProtection OID or an ExtendedKeyUseAny OID. These are not mutually exclusive and other EKU OIDs may appear as well.

Apple iChat certs must contain at least one of three EKU OIDs:

- CSSMOID_ExtendedKeyUsageAny
- CSSMOID_APPLE_EKU_ICHAT_SIGNING
- CSSMOID_APPLE_EKU_ICHAT_ENCRYPTION

The app can **require** the presence of either or both of the last two EKU OIDs by asserting appropriate bits in the CSSM_APPLE_TP_SMIME_OPTIONS.IntendedUsage field. These bits are defined in <Security/certextensions.h>, along with the typedef for CE_KeyUsage. These bits are normally used in the KeyUsage extension (which is in fact checked for S/MIME cert evaluation, which shares this data structure). Here only two bits are legal:

- CE_KU_DigitalSignature. If this bit is set then the cert **must** contain an ICHAT_SIGNING EKU OID.
- CE_KU_DataEncipherment. If this bit is set then the cert **must** contain an ICHAT_ENCRYPTION EKU OID.

If the app does not provide an IntendedUsage value (i.e., it's zero) then any of the 3 possible EKU OIDs will suffice for the satisfaction of the ExtendedKeyUse verification.

A failure of the extended key use extension verification results in a CSSMERR_APPLETP_SMIME_BAD_EXT_KEY_USE error.

Note that if the app specifies a nonzero IntendedUsage value, but the cert being evaluated is a generic AIM cert, the IntendedUsage field will be ignored since the generic AIM cert does not contain indicators as to what it is intended to be used for.

CodeSigning

Certificate chains used in Apple Code Signing consist of exactly three certificates: the Apple Root Certificate (ARC), an intermediate certificate, and end entity (leaf) certificate. If there are not 3 certificates in the chain being evaluated, CSSMERR_APPLETP_CS_BAD_CERT_CHAIN_LENGTH is returned. The intermediate cert must have a BasicConstraints extension; if it doesn't, CSSMERR_APPLETP_CS_NO_BASIC_CONSTRAINTS is returned. That BasicConstraints must have PathLengthConstraint of zero. If it doesn't, CSSMERR_APPLETP_CS_BAD_PATH_LENGTH is returned. A value of zero for the PathLengthConstraint field indicates that no more intermediate certificates are allowed in the cert chain; the next certificate must be the leaf.

The single intermediate certificate must have an ExtendedKeyUse extension; if it doesn't, CSSMERR_APPLETP_CS_NO_EXTENDED_KEY_USAGE is returned. That extension must be marked critical; and that extension must have exactly one of two values: appleCodeSigning, or appleCodeSigningDevelopment. The presence of the latter value is considered an exception condition and will result in a CSSMERR_APPLETP_CODE_SIGN_DEVELOPMENT being returned. If the EKU extension does not have exactly one value present, CSSMERR_APPLETP_INVALID_EXTENDED_KEY_USAGE is returned.

2.4.2 Revocation policies

Revocation Policies refer to those policies implemented by the TP which (optionally) check whether the certificates in a given cert chain have been revoked by their issuer. There are currently two revocation policies: Certificate Revocation List (CRL) and Online Certificate Status Protocol (OCSP). Both of these policies optionally involve fetching items from the Internet during the verification of a cert; both involve caching of said items both in the TP (in-memory, on a per-process basis) and elsewhere in the system (on disk, in a cache shared by all users); both involve per-user preferences set at the User Interface level (by the Keychain Access application); both have somewhat "special case" status in the implementation of the TP's main "customer" code, the SecTrust layer. None of these issues will be discussed here expect for brief mentions as to the options, at the TP API, associated with network access and caching.

The two policies are almost, but not totally, independent. Either, both, or neither of the two policies can be enabled for a given cert chain verification. However there is one bit in the top-level CSSM_APPLE_TP_ACTION_DATA, the REQUIRE_REV_PER_CERT bit, which when set, results in the following interdependency between the CRL and OCSP policies: for each cert in the cert chain being verified, a positive confirmation of a cert's revocation status must be obtained by **either** the CRL or OCSP policy, even if either or both policies are configured to **not** require positive confirmation per cert. (A "positive confirmation" means that a definitive "this cert has not been revoked" status has been obtained, not just a "I couldn't contact the server" status, which may or may not be an error on a per-policy basis.)

Asserting the REQUIRE_REV_PER_CERT bit does not cause either revocation policy to be executed if the caller has not told the TP to execute the policy; for example, suppose the caller has enabled CRL checking but not OCSP, and the CRL policy is configured to not actually fail if no CRL can be obtained for a given cert. In this case, if REQUIRE_REV_PER_CERT is set, the verification returns CSSMERR_APPLETP_INCOMPLETE_REVOCATION_CHECK.

CRL policy

The CRL policy attempts to locate a current CRL for each cert, to cryptographically verify the CRL and each cert in the chain that verifies it, perform CRL verification of each of those certs, and finally to determine if the CRL lists the target cert in question as being revoked. CRLs can come from a number of places. The TP searches for them in the following places, listed in the order in which they are searched:

- The caller can explicitly provide them in CSSM_TP_VERIFY_CONTEXT.Crls
- The caller can provide them in a DLDB list, in CSSM_TP_CALLERAUTH_CONTEXT.DBList
- A process-wide memory-resident cache
- A system-wide disk-resident cache
- They can be obtained from the net, if so enabled (by the FETCH_CRL_FROM_NET bit, see below) and the cert contains a CrlDistributionPoints extension with the URL specified FullName form (not in relative form)

The CRL policy takes an optional pointer to a policy-specific struct, CSSM_APPLE_TP_CRL_OPTIONS, in VerifyContext->Cred->Policy.PolicyIds->FieldValue.Data. The fields in this struct are:

Version: must be CSSM_APPLE_TP_CRL_OPTS_VERSION

CrlFlags: See below.

crlStore: currently unused.

The values for the bits in CrlFlags is as follows. Note that the behavior of the CRL policy if no CSSM_APPLE_TP_CRL_OPTIONS is present is such that all of these bits are assumed to be zero.

CSSM_TP_ACTION_REQUIRE_CRL_PER_CERT

If set, a valid CRL must be located for each cert. If not, the verification will fail with the reason for the most recent CRL attempt failed (typically, CSSMERR_APPLETP_CRL_NOT_FOUND, but it could be a verification failure for the most recent CRL fetched).

CSSM_TP_ACTION_FETCH_CRL_FROM_NET

If set, the TP will attempt to fetch CRLs from the internet if the cert contains a CrlDistributionPoints extension with the URL specified FullName form. (Note this will not occur if a CRL is found for a particular cert in cache or in a caller-provided location).

CSSM_TP_ACTION_CRL_SUFFICIENT

If set, then revocation checking by other policies (OCSP for now) is not performed for certs which have been positively verified by CRL checking, regardless of the configuration of the other revocation policies. The caller sets this to indicate "If CRL checking is successful for a given cert, that's good enough for me; don't bother with other revocation checking."

CSSM_TP_ACTION_REQUIRE_CRL_IF_PRESENT

This is identical to the REQUIRE_CRL_PER_CERT bit except it only applies to certs which have a CrlDistributionPoints extension. It basically means "if the issuer says this cert has a CRL then you must get positive CRL verification, otherwise tolerate the inability to find a CRL for the cert."

OCSP policy

OCSP is a more real-time policy than CRL; the basic intent is that a server which is authorized by the issuer of a cert being verified is to be queried at the time the cert is being verified; the server (actually called a "responder" in this context) will give a response indicating the revocation status of the cert. Of course it's not quite that simple; responses from OCSP servers are cached for performance reasons and to guard against situations when, e.g., your laptop is not connected to the network.

OCSP responses (the things that come from a responder indicating the revocation status of the cert being verified) are themselves cryptographically signed; the TP verifies an OCSP response before it's used. (The in-memory OCSP cache contains OCSP responses which are verified once and then explicitly trusted. These responses' thisUpdate and nextUpdate times are still honored as usual.) Each cert in the cert chain used to verify an OCSP response is in turn subjected to OCSP verification.

The OCSP policy takes an optional pointer to a policy-specific struct, CSSM_APPLE_TP_OCSP_OPTIONS, in VerifyContext->Cred->Policy.PolicyIds->FieldValue.Data. This struct is currently defined in <Security/cssmapplePriv.h>. The fields in this struct are:

Version: must be CSSM_APPLE_TP_OCSP_OPTS_VERSION

Flags: See below.

LocalResponder: Optional URI of a local OCSP responder to use instead of any possible responder indicated in certs.

LocalResponderCert: Optional certificate associated with LocalResponder, to provide intergrity checking of the connection to LocalResponder.

The values for the bits in Flags is as follows. Note that the behavior of the OCSP policy if no CSSM_APPLE_TP_OCSP_OPTIONS is present is such that all of these bits are assumed to be zero.

CSSM_TP_ACTION_REQUIRE_OCSP_PER_CERT

If set, a valid OCSP response must be located for each cert. If not, the verification will fail with the reason for the most recent OCSP attempt failed (typically, CSSMERR_APPLETP_OCSP_UNAVAILABLE, but it could be a verification failure for the most recent OCSP response obtained).

CSSM_TP_ACTION_OCSP_REQUIRE_IF_RESP_PRESENT

This is identical to the REQUIRE_OCSP_PER_CERT bit except it only applies to certs which have an AuthorityInfoAccess extension with an id-ad-ocsp URI. It basically means "if the issuer says it provides OCSP for this cert, then you must get positive OCSP verification, otherwise tolerate the inability to find an OCSP responder (or response) for the cert."

CSSM_TP_ACTION_OCSP_DISABLE_NET

If set, no network transactions will be attempted in order to perform OCSP verification. The default is that the network will be used.

CSSM_TP_ACTION_OCSP_CACHE_READ_DISABLE
CSSM_TP_ACTION_OCSP_CACHE_WRITE_DISABLE

The control the use of local cache (both memory and disk resident). The default is that OCSP responses will be written to cache and can be fetched from cache. Either reading from or writing to cache can be disabled.

CSSM_TP_ACTION_OCSP_SUFFICIENT

If set, then revocation checking by other policies (CRL for now) is not performed for certs which have been positively verified by OCSP, regardless of the configuration of the other revocation policies. The caller sets this to indicate "If OCSP is successful for a given cert, that's good enough for me; don't bother with other revocation checking."

CSSM_TP_OCSP_GEN_NONCE
CSSM_TP_OCSP_REQUIRE_RESP_NONCE

These control the generation of nonces in client-side OCSP requests, and whether servers must reply with nonces in their responses. Default is false for both.

3.0 Revision History

Rev	Date	Change
0.1	5/6/99	Creation.
0.2	5/7/99	Added CSSM_TP_VERIFY_ACTION_FAIL return code.
		Added this Appendix.
0.3	….	Changed description of defaults for basicConstraints.cA.
		Added CSSM_TP_INVALID_CERTIFICATE return code**.**
		Clarified description of fully successful return.
0.4	7/28/99	Fixed misc. typos.
		Changed Default policy to AppleBasic
		Added CSSM_TP_CERT_EXPIRED.
		Added description of verifying root cert against known root certs.
		Added Netscape exception to iSign's Key Usage requirements.
		Added section 4.2, description of Embedded Root Certs.
		Removed second paragraph in 1.0 containing cynical disclaimer regarding policy changes.
0.9	8/20/01	Update for Mac OS X.
0.91	10/18/01	Fixed some typos.
1.00	8/14/02	Major revision to CSSM_TP_CertGroupVerify description.
		Minor updates of per-policy extension parsing.
1.1	1/25/05	Major update for Tiger.
		Several new policies.
		New description of revocation policies.