High Level PKI Services API
3/20/2002

This document specifies a high-level Application Programming Interface (API) for public-key based cryptographic services. Currently, PKI-enabled applications must use proprietary, vendor-provided APIs to interface with their PKI, thus making support across multiple PKI products difficult. To facilitate the development and wide-deployment of PKI-enabled applications, NIST is working with several federal agencies to make this interface to a PKI consistent, regardless of the PKI product being used. If each PKI product and each application can meet at a common interface, more applications will become PKI enabled for all PKIs. Figure 1 illustrates the components of a PKI-enabled application and the specific interface that this document attempts to address.

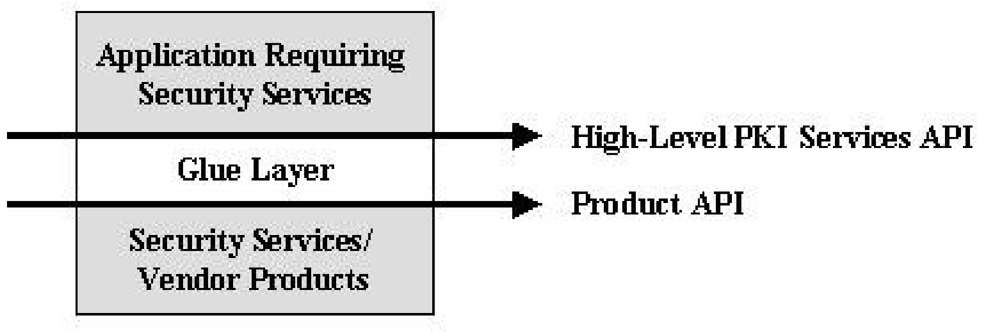
Application Requiring Security Services
Glue Layer
Security Services/ Vendor Products
High-Level PKI Services API
Product API

Figure 1. High-Level PKI Services API

The **application requiring security services** is any application that needs digital signature and/or encryption services. The **security services/vendor products** are the existing vendor products that provide the signing and encryption functionality. The **product API** is the vendor-specific interface provided by the product for calling the signing and encryption services. The **high-level PKI services API** is the common API that this document specifies to provide a consistent interface to signing and encryption services irrespective of the product being used. The high-level API is designed to hide the complexity of the underlying security mechanisms but facilitate service requests through simple service calls. The **glue layer** is the code necessary to translate the high-level PKI services API into the product API.

In this specification, the term "PKI" is loosely used to refer to all the components below the high-level API. Several assumptions are made with respect to this API specification. They are:

1. All calls are made on behalf of one user at a time.

2. This API is designed for an authentication framework where end entity certificates are used for authentication purpose only. The specific roles that a user is authorized to play is handled by the applications, and the use of attribute certificates is out of scope of this API.

3. A PKI-enabled application may use a single login process for a user to login to the application and the PKI. If a user is not logged in when a PKI service is invoked, the user will be prompted to login. Once logged in, the user's PKI identity is maintained, possibly through a context object opaque to the application but accessible to the PKI, until the user logs out of the PKI or exits the application. Therefore, it is unnecessary for the application to pass the user's identity each time a cryptographic function is called. A PKI can provide a mechanism

for a user to log in and out of the PKI independent from the application. Implementers of the API should be aware of this fact and always make sure that a user is logged in before a PKI service can be rendered to the user. The PKI should allow a user to reauthenticate himself after a timeout from inactivity. Certain calls in this API include a parameter *authent_required* to enforce reauthentication for applications that require specific high level of assurance.

4. The PKI can be preconfigured to contain certain information that is needed by most PKI-enabled applications. Information such as the location of the certificate repository, the default signing, encryption algorithms, the policy or assurance level required by certain applications, can be summarized in a configuration file, and accessed, enforced during the provision of the PKI services. A configuration file may be generated automatically by a PKI product when a user registers with a Certification Authority and obtains his certificate. The PKI can provide tools such as address book to help user search for needed certificates. Similarly, policy or assurance levels such as BASIC, LOW, MEDIUM, HIGH or 0, 1, 2, 3 can be mapped to registered policy OIDs in the configuration file. It is assumed that a higher policy number implies a higher assurance level.

   X.509 allows the specification of an initial policy set as well as two flags - one inhibiting policy mapping, and the other requiring policies to be present in all certificates in the path. These settings can be used to constrain certificate path processing. The API does not expose these settings. The system administrator should configure these settings when the CA's certificate or the end entity’s certificate is generated, and perhaps document such settings in the configuration file.

5. For signature generation, only one piece of information will be signed at a time. This decision is made from the security point of view. Signing information should be a conscious effort where a user knows exactly what he is signing. It is therefore preferable that only one piece of information is signed at a time.

6. When a digital signature is generated, the DER-encoded signature octet string is encapsulated in the SignedData type of RFC 2630 [1]. The original signed content can be encapsulated in the EncapsulatedContentInfo of the SignedData type as is the case with “opaque signing”, but this inclusion is optional (RFC 2630, section 5.2). In the case of “clear signing” where the signed content is not encapsulated, the application will have to pass the signed content to the API for signature verification. **And it is the application’s responsibility to properly manage the association of the signed content and the associated signature.** Both clear signing and opaque signing have merits for certain applications. The technical and legal ramifications of using clear signing vs. opaque signing are beyond the scope of this specification, and are more suitable for inclusion in an implementation guidance document. Nonetheless, an application should choose between the two based on its specific need and must be vigilant about maintaining the integrity of the data that is signed and its signature.

7. For signature verification, it is assumed that a receiving party will receive enough information that is needed to verify a signature. However, the receiving party is not expected to receive all the certificates and CRLs that are needed for certificate path validation. The receiving party’s PKI is responsible for such path construction and validation in the process of signature verification.

8. For encryption, a piece of information may be encrypted for multiple recipients with one call using the same encryption algorithm. The encryption and key management algorithms, and

the location of the certificate repository…etc., can be preconfigured in a file and modified as needed.

When a piece of information is encrypted, the cipher text (optional) and other relevant information needed for decryption is DER-encoded and encapsulated in the EnvelopedData type of RFC 2630. The cipher text can be omitted from the EncryptedContentInfo field of the EnvelopedData type (RFC 2630, section 6.1). In this case, the cipher text must be supplied by other means. To avoid the additional data management for the application, it is assumed in this API that the cipher text will be included in the EncryptedContentInfo.

An implementation to support the API may provide a tool to help users search for end entity certificates and construct the recipient list for encrypted messages. Depending on what is expected by the underneath vendor-specific encryption function, this tool may guide the user to provide information suitable for identifying an intended recipient such as a name or an email address. The tool will use such information to search the repository for the desired certificates and construct the recipient list needed by the encryption function. The sender of an encrypted message may not be automatically on the recipient list. Therefore, if the sender wishes to receive a copy of the cipher text, his name should be added to the list.

9. If the application wishes to encrypt and sign the same piece of data, RFC 2630 does not require the two procedures to be performed in a specific order. The application may receive encapsulated data that has been signed and/or encrypted in any order.

   Note:

   For FDIC PKI applications, if an object needs to be signed and encrypted, it is recommended that the object be signed first, then encrypted, then optionally resigned. The rationale being that if an object is encrypted then signed, it will be difficult to establish technical non-repudiation. The optional resigning can be used to prevent denial of service attack; if the outer signature does not verify then there is no need to decrypt the data and verify the inner signature. If the data is only signed then encrypted, the application must decrypt the data and attempt to verify the inner signature. The downside of sign-encrypt-resign is that processing authentic data will require two signature verifications and one decryption.

10. It is assumed that the application will allocate and deallocate the memory for all the input and output parameters. For output that has fixed size or relatively small size, the application should simply allocate a reasonable amount of memory for such parameter. In cases where the length of the output is more difficult to predict, the application should supply a length parameter along with the actual output. As input to the API, this length parameter specifies the size of memory, in bytes, that has been allocated for the associated output. If the PKI determines that more space is needed, the API will return INSUFFICIENT_BUFFER_SIZE as the error code and indicate the actual size needed in the length parameter. Based on this information, the application should free up the space previously allocated and reallocate the needed space before calling the specific function again. Alternatively, the application can allocate nothing and specify zero for the length and NULL for the associated output on the first attempt, wait for returned information from the API, then make the necessary allocation and call the function again. The API developers should check the incoming length and the associated buffer. If the length comes in as zero and/or the buffer comes in as a NULL pointer, the API should return INSUFFICIENT_BUFFER_SIZE and return the amount of space that is needed in the length parameter. When the application receives an

INSUFFICIENT_BUFFER_SIZE error code, other than the length and status message, it should not assume any valid information would be contained in the other output parameters.

11. Two sets of cryptographic functions are specified in this API: one set for file oriented operations, and another for buffer oriented operations. Each function will return a return code. If there is no error encountered, the function will return 0; otherwise it will return an error code listed in the Appendix. Additional information may be returned in *error_data* for debugging purpose.

    For a glue-layer implementation that maps the high-level API to low-level vendor-specific cryptographic API, the glue-layer must also map the vendor-specific error codes to the return codes specified in the Appendix. This will allow an application to handle error codes from different PKI implementations. A unique code ERR_UNKNOWN is defined for vendor specific error codes that do not correspond to any other error code specified in the Appendix. In this case, it is recommended that the vendor specific error code, plus error description, be included in *error_data* for tracking purpose.

    In describing the following functions, the input and output parameters are listed along with specific data types for the arguments. Data types such as SignedData and EnvelopedData are DER-encoded octet strings that are defined in RFC 2630. GeneralizedTime is an ASN.1 [2] data type used to represent time and date in the YYYYMMDDHHMMSS.SSS form with some permitted variations.

High Level PKI Service Calls

```
int signBuffer(
    IN        uint32        data_length,
    IN        char*         data_to_sign,
    IN        Boolean       authent_required,
    IN        Boolean       encap_data_flag,
    IN/OUT    uint32*       signed_data_length,
    IN/OUT    SignedData*   signed_data,
    OUT       char*         error_data
);
```

This function will cause the underlying components to
- Reauthenticate the user to the PKI if the *authent_required* flag is set to TRUE.
- Locate the signer's key and generate digital signature over the data to be signed.
- DER-encode the signature and relevant information (such as signerInfo) in the SignedData type of RFC2630.
- Return error code or success to the application.

signBuffer() arguments:
data_length: the length of *data_to_sign*.
data_to_sign: the buffer for the data to be signed.
authent_required: Specifies whether a user should be reauthenticated to the PKI before a digital signature can be generated.
encap_data_flag: Specifies whether the signed content should be included in EncapsulatedContentInfo.

signed_data_length: A pointer to the size of the *signed_data* buffer. As input to the API, it points to the memory size allocated for *signed_data*. As output, it points to the actual length of *signed_data*.
signed_data: the buffer to hold the signature octet string.
error_data: the buffer to hold pertinent information about the error condition.

<u>signBuffer() return values:</u> See assumption 11.

```
int signFile(
    IN        char*          infile,
    IN        Boolean        authent_required,
    IN        Boolean        encap_data_flag,
    IN        Boolean        output_to_file,
    IN        char*          outfile,
    IN/OUT    uint32*        signed_data_length,
    IN/OUT    SignedData*    signed_data,
    OUT       char*          error_data
);
```

This function will cause the underlying components to

- Reauthenticate the user to the PKI if the *authent_required* flag is set to TRUE.
- Open and read the file to be signed; return error if it can not be opened or read.
- Locate the signer's key and generate a digital signature over the file to be signed.
- DER-encode the signature and other relevant information (such as signerInfo) in the SignedData type. Write the signature output to the *signed_data* buffer or to the *outfile* file, depending on the choice specified in *output_to_file*.
- Return error code or success to the application.

<u>signFile() arguments:</u>
infile: the name of the file to be signed.
authent_required: Specifies whether a user should be reauthenticated to the PKI before a digital signature can be generated.
encap_data_flag: Specifies whether the file to be signed should be included in EncapsulatedContentInfo of SignedData. The default is FALSE. It is generally undesirable to include the signed file, unless small in size, in EncapsulatedContentInfo.
output_to_file: Specifies whether the signature output should be sent to a file or a buffer. If the output is sent to a buffer, *signed_data_length* and *signed_data* shall be used; and *verifyBuffer* rather than *verifyFile* should be called to verify the signature contained in the buffer. If the signature output is to be written to *outfile*, then *verifyFile* should be called later on for signature verification.
outfile: the name of the file to receive the signature output.
signed_data_length: A pointer to the size of the *signed_data* buffer. As input to the API, it points to the memory size allocated for *signed_data*. As output, it points to the actual length of *signed_data*. *Signed_data_length* and *signed_data* shall be used only when *output_to_file* is FALSE.
signed_data: the buffer to hold the signature octet string. *Signed_data_length* and *signed_data* shall be used only when *output_to_file* is FALSE.
error_data: the buffer to hold pertinent information about the error condition.

<u>signFile() return values:</u> See assumption 11.

```
int verifyBuffer(
    IN        uint32              signed_data_length,
    IN        SignedData*         signed_data,
    IN        ushort              policy,
    OUT       char*               signer,
    OUT       GeneralizedTime     time_data_signed,
    IN/OUT    uint32*             data_verified_length,
    IN/OUT    char*               data_verified,
    OUT       char*               error_data
);
```

This function is used to verify the signature contained in the *signed_data* buffer. Upon a successful verification, the signer identity, timestamp, and the original signed content are returned.

verifyBuffer() arguments:
signed_data_length: the length of *signed_data*.
signed_data: the buffer that holds the signature octet string
policy: the required policy or assurance level under which the verification certificate must be issued. Applications that do not require policy checking can set this to zero as input.
signer: the buffer to receive the signer identity.
time_data_signed: the buffer to receive the signing date and time.
data_verified_length: Specifies the length of *data_verified,* which is also the data signed. Depending on whether the signed content was included in EncapsulatedContentInfo, this parameter has different settings as input to the API. If the signed content was included in EncapsulatedContentInfo, this parameter, as input to the API, points to the memory size allocated for *data_verified*. As output, it points to the actual length of *data_verified*. However, if the original signed content was not included in EncapsulatedContentInfo, then it should be specified through *data_verified* and *data_verified_length* for signature verification. In this case, as input and output to the API, this parameter always points to the length of the signed content held in the *data_verified* buffer.
data_verified: the buffer to hold or holding the signed content depending on whether the content was included in EncapsulatedContentInfo. If the signed content was included in EncapsulatedContentInfo, this parameter, as input to the API, specifies an empty buffer. As output, it holds the signed content whose signature is just verified. On the other hand, if the signed content was not included in EncapsulatedContentInfo, this parameter, as input and output to the API, buffers the data that was signed.
error_data: the buffer to hold pertinent information about the error condition.

verifyBuffer() return values: See assumption 11.

```
int verifyFile(
    IN/OUT    char*               file_signed,
    IN        char*               signature_file,
    IN        ushort              policy,
    OUT       char*               signer,
    OUT       GeneralizedTime     time_data_signed,
    OUT       char*               error_data
);
```

This function is used to verify a digital signature contained in the *signature_file*. Upon a successful verification, the signer identity, timestamp, and the original signed content are returned.

<u>verifyFile() arguments:</u>

file_signed: names the file to receive or holding the signed data depending on whether the original signed content was included in EncapsulatedContentInfo. If the signed file content was included in EncapsulatedContentInfo, this parameter should specify a file name to receive the content signed upon a successful verification. If the file content was not included in EncapsulatedContentInfo, this parameter, as input and output to the API, names the file that was signed.
signature_file: names the file that contains the signature octet string.
policy: the required policy or assurance level under which the verification certificate must be issued. Applications that do not require policy checking can set this to zero as input.
signer: the buffer to receive the signer identity.
time_data_signed: the buffer to receive the signing date and time.
error_data: the buffer to hold pertinent information about the error condition.

<u>verifyFile() return values:</u> See assumption 11.

```
int encryptBuffer(
    IN        uint32            data_length,
    IN        char*             data_to_encrypt,
    IN        ushort            encryption_algorithm,
    IN        Boolean           authent_required,
    IN/OUT    uint32*           enveloped_data_length,
    IN/OUT    EnvelopedData*    enveloped_data,
    OUT       char*             error_data
);
```

This function will cause the underlying components to

- Construct the recipient list and locate each recipient's encryption certificate or key agreement public key certificate depending on the choice of key management techniques. Check the validity of the recipient's certificate. Locate the sender's private key if key agreement is selected.
- Generate a random session or one-time symmetric key. If key transport is the choice for deriving the same encrypting key between the sender and the recipient, then the session or one-time key should be encrypted under the recipient's public encryption key. If key agreement is the choice, then the sender's private key and the recipient's key agreement public key are used to generate a pair wise symmetric key, which is then used to encrypt the session or one-time key. If the key management choice is neither key transport nor key agreement, but rather to use a previously distributed symmetric key encryption key, then the session or one-time key should be encrypted under this key encryption key. In order for this key management scheme to work, the sender must have a previously distributed key encryption key with each recipient that the sender may communicate with. Since this key management scheme does not scale well to a large user community, we recommend that only key transport or key agreement be used.
- Encrypt the data buffer under the session or one-time key using the encryption algorithm specified in *encrytion_algorithm.*

- DER-encode the recipient-specific information and cipher text in the envelopedData type of RFC2630. Note that the application will send the same DER-encoded envelopedData to all the recipients. It is the responsibility of each recipient's PKI to decode the envelopedData and parse the recipient-specific information in order to derive the encryption key that is to decrypt the cipher text.

encryptBuffer() arguments
data_length: the length of the *data_to_encrypt* buffer.
data_to_encrypt: the data buffer to be encrypted.
encryption_algorithm: the encryption algorithm to be used.
authent_required: Relevant only if key agreement is used during the encryption process, this flag specifies whether the user needs to be reauthenticated to the PKI before the private key can be used to perform key agreement. If key agreement is not used, this flag should be set to FALSE.
enveloped_data_length: A pointer to the size of the *enveloped_data* buffer. As input to the API, it points to the memory size allocated for *enveloped_data*. As output, it points to the actual size of *enveloped_data*.
enveloped_data: the buffer to hold the encrypted octet string that corresponds to the EnvelopedData type of RFC 2630.
error_data: the buffer to hold pertinent information about the error condition.

encryptBuffer() return values: See assumption 11.

```
int encryptFile(
    IN        char*            file_to_encrypt,
    IN        ushort           encryption_algorithm,
    IN        Boolean          authent_required,
    IN        Boolean          output_to_file,
    IN        char*            encrypted_file,
    IN/OUT    uint32*          enveloped_data_length,
    IN/OUT    EnvelopedData*   enveloped_data,
    OUT       char*            error_data
);
```

This function will cause the underlying components to

- Open the specific files for reading/writing. Return error code if it encounters a problem.
- Construct the recipient list and locate each recipient's encryption certificate or key agreement public key certificate depending on the choice of key management techniques. Check the validity of the recipient's certificate. Locate the sender's private key if key agreement is selected; the sender may need to be reauthenticated if *authent_required* is set to TRUE.
- Generate a random session or one-time symmetric key. If key transport is the choice for deriving the same encrypting key between the sender and the recipient, then the session or one-time key should be encrypted under the recipient's public encryption key. If key agreement is the choice, then the sender's private key and the recipient's key agreement public key are used to generate a pair wise symmetric key, which is then used to encrypt the session or one-time key. If the key management choice is neither key transport nor key agreement, but rather to use a previously distributed symmetric key encryption key, then the session or one-time key should be encrypted under this key encryption key. In order for this key management scheme to work, the sender must have a previously distributed key encryption key with each recipient that the sender may communicate with. Since this key

management scheme does not scale well to a large user community, we recommend that only key transport or key agreement be used.

- Read the file content into a data buffer; encrypt the buffer under the session or one-time key using the specified encryption algorithm.
- DER-encode the recipient-specific information and cipher text in the envelopedData type of RFC2630. Depending on the choice specified in *output_to_file*, the encrypted output should be written to a buffer, *enveloped_data,* or to a file named by *encrypted_file*. Note that the application will send the same encrypted file or buffer to all the recipients. It is the responsibility of each recipient's PKI to read the file or buffer, decode the envelopedData, and parse the recipient-specific information in order to derive the encryption key that is to decrypt the cipher text.

encryptFile() arguments

file_to_encrypt: the name of the file to be encrypted.

encryption_algorithm: the encryption algorithm to be used.

authent_required: Relevant only if key agreement is used during the encryption process, this flag specifies whether the user needs to be reauthenticated to the PKI before the private key can be used to perform key agreement. If key agreement is not used, this flag should be set to FALSE.

output_to_file: Specifies whether the encrypted output should be sent to a file or a buffer. If the output is sent to a buffer, *enveloped_data_length* and *enveloped_data* shall be used; and *decryptBuffer* rather than *decryptFile* should be called to decrypt the encrypted buffer. If the output is sent to the *encrypted_file* file, then *decryptFile* should be called to decrypt the encrypted file.

encrypted_file: names the file to receive the encrypted output.

enveloped_data_length: A pointer to the size of the *enveloped_data* buffer. As input to the API, it points to the memory size allocated for *enveloped_data*. As output, it points to the actual length of *enveloped_data*. This parameter and *enveloped_data* shall be used only when *output_to_file* is set to FALSE.

enveloped_data: the buffer to hold the encrypted octet string. This parameter and *enveloped_data_length* shall be used only when *output_to_file* is set to FALSE.

error_data: the buffer to hold pertinent information about the error condition.

encryptFile() return values: See assumption 11.

```
int decryptBuffer(
    IN        uint32            enveloped_data_length,
    IN        EnvelopedData*    enveloped_data,
    IN        Boolean           authent_required,
    IN/OUT    uint32*           plain_text_length,
    IN/OUT    char*             plain_text,
    OUT       ushort*           encryption_algorithm,
    OUT       char*             error_data
);
```

This function is used to decrypt an encrypted *enveloped_data* buffer. Upon successful decryption, the plain text and the encryption algorithm used are returned.

decryptBuffer() arguments

enveloped_data_length: the length of the *enveloped_data* buffer.

enveloped_data: the buffer that holds the encrypted octet string.

authent_required: Relevant only if key transport or key agreement is used during the encryption process, this flag specifies whether the user needs to be reauthenticated to the PKI before his private key can be used to perform key transport or key agreement. If key transport or key agreement is not used, this flag should be set to FALSE.
plain_text_length: a pointer to the size of the *plain_text* buffer. As input to the API, it points to the memory size allocated for *plain_text*. As output, it points to the actual length of *plain_text*.
plain_text: the buffer to receive the decrypted text.
encryption_algorithm: a pointer to the encryption algorithm used.
error_data: the buffer to hold pertinent information about the error condition.

decryptBuffer() return values: See assumption 11.

```
int decryptFile(
    IN        char*            encrypted_file,
    IN        Boolean          authent_required,
    IN        char*            plain_text_file,
    OUT       ushort*          encryption_algorithm,
    OUT       char*            error_data
);
```

This function is used to decrypt an encrypted file. Upon successful decryption, the plain text and the encryption algorithm used are returned.

decryptFile() arguments
encrypted_file: the name of the file that contains the encrypted octet string.
authent_required: Relevant only if key transport or key agreement is used during the encryption process, this flag specifies whether the user needs to be reauthenticated to the PKI before his private key can be used to perform key transport or key agreement. If key transport or key agreement is not used, this flag should be set to FALSE.
plain_text_file: the name of the file that is to receive the decrypted text.
encryption _algorithm: a pointer to the encryption algorithm used.
error_data: the buffer to hold pertinent information about the error condition.

decryptFile() return values: See assumption 11.

```
int CMSBufferParser (
    IN        uint32           signed_data_length,
    IN        SignedData*      signed_data,
    OUT       char*            signer,
    OUT       GeneralizedTime  time_data_signed,
    IN/OUT    uint32*          content_length,
    IN/OUT    char*            content_signed,
    OUT       char*            error_data
);
```

*** Note this is a non-cryptographic function call.** It allows an application that is unaware of the complex CMS structure to be able to obtain information about the signer without having to verify the signature. The idea came from the S/MIME (Secure/Multipurpose Internet Mail Extensions) client implementation where one may receive a signed message but does not feel the need to verify the signature, and yet wants to know what was signed and who signed it. Note that if the content that was signed had not been included in the EncapsulatedContentInfo of SignedData, then the output *content_length* should return zero and *content_signed* shall be an empty string.

<u>CMSBufferParser() arguments:</u>
signed_data_length: the length of the *signed_data* buffer.
signed_data: the buffer that holds the signature octet string.
signer: the buffer to receive the signer identity.
time_data_signed: the buffer to receive the signing date and time.
content_length: a pointer to the size of *content_signed*. As input to the API, it points to the memory size allocated for *content_signed*. As output, it points to the actual length of *content_signed*.
content_signed: the buffer to receive the original signed content. If that content was not included in EncapsulatedContentInfo, then *content_signed* should be an empty string.
error_data: the buffer to hold pertinent information about the error condition.

<u>CMSBufferParser() return values:</u> See assumption 11.

```
int CMSFileParser (
    IN          char*               signature_file,
    IN/OUT      char*               file_signed,
    OUT         char*               signer,
    OUT         GeneralizedTime     time_data_signed,
    OUT         char*               error_data
);
```

- **Note this is a non-cryptographic function call.** It allows an application that is unaware of the complex CMS structure to be able to obtain information about the signer without having to verify the signature. The idea came from the S/MIME client implementation where one may receive a signed message but does not feel the need to verify the signature, and yet wants to know what was signed and who signed it. In the case of a signed file, the file content by default will not be included in the EncapsulatedContentInfo, therefore, the signed content will not be written to *file_signed*. However, if the signed content was included in EncapsulatedContentInfo, the signed content will be written to the *file_signed* file.

<u>CMSFileParser() arguments:</u>
signature_file: the name of the signature file.
file_signed: the name of the file to receive the original signed content.
signer: the buffer to receive the signer identity.
time_data_signed: the buffer to receive the signing date and time.
error_data: the buffer to hold pertinent information about the error condition.

<u>CMSFileParser() return values:</u> See assumption 11.

APPENDIX – ERROR CODES

Error Code	**Meaning**	**If present, data is**
ERR_INVALID_PARAMETER	Parameter does not contain required information	Parameter in error
ERR_INSUFFICIENT_BUFFER_SIZE	Allocated memory is not sufficient	Parameter in error
ERR_ALLOC_ERROR	Could not allocate memory	Parameter in error
ERR_FILE_OPERATION	Error occurred in a file operation	Filename
ERR_KEY_NOT_FOUND	The required key could not be found	DN of key owner
ERR_UNSUPPORTED_ALGORITHM	The algorithm in this protected object is unsupported	Algorithm ID
ERR_ASN1_PARSE_FAILURE	Could not parse an ASN.1 object	
ERR_CERT_INVALID	The certificate is not valid	DN of certificate
ERR_CERT_PATH_ERROR	No valid path can be found to validate a certificate	DN of certificate
ERR_CERT_EXPIRED	The certificate is expired	DN of certificate
ERR_CERT_ASSURANCE_LEVEL_NOT_MET	Certificate policy does not meet the required assurance level	DN of certificate
ERR_CERT_EXT_UNKNOWN_CRITICAL	The certificate contains an unknown extension marked critical	DN of certificate
ERR_RR_UNSPECIFIED	The certificate was revoked with an unspecified reason	DN of revoked certificate
ERR_RR_KEY_COMPROMISE	The certificate was revoked with the reason of key compromise	DN of revoked certificate
ERR_RR_AFFILIATION_CHANGED	The certificate was revoked with the reason of affiliation changed	DN of revoked certificate
ERR_RR_SUPERSEDED	The certificate was revoked with the reason of superseded	DN of revoked certificate
ERR_RR_CESSATION_OF_OPERATION	The certificate was revoked with the reason of cessation of operation	DN of revoked certificate
ERR_RR_OTHER	The certificate was revoked	DN of revoked certificate
ERR_CRL_NOT_FOUND	Could not find the required CRL	DN of CRL issuer
ERR_CRL_SIGNATURE_FAILED	The signature on a CRL could not be verified	DN of CRL issuer
ERR_CRL_EXT_UNKNOWN_CRITICAL	The CRL contains an unknown extension marked critical	DN of CRL issuer
ERR_WRONG_CRYPTO_SEQUENCE	Wrong sequence of cryptographic function is called	
ERR_SIG_VERIFY_FAILED	Failed to verify a signature	
ERR_ENCRYPT_FAILED	Failed to encrypt data	
ERR_DECRYPT_FAILED	Failed to decrypt data	
ERR_UNKNOWN	Failed service or resource unavailable	Error description received from underneath PKI service calls

References

[1] Housley, R. Cryptographic Message Syntax. RFC 2630. June 1999.

[2] ITU-T X.680 : OSI networking and system aspects – Abstract Syntax Notation One (ASN.1). December 1997.