

;

Digital Signatures and PKCS#11 Smart Cards

Concepts, Issues and some Programming Details



CALIFORNIA SOFTWARE LABS

REALIZE YOUR IDEAS

California Software Labs

6800 Koll Center Parkway,

Suite 100 Pleasanton

CA 94566, USA.

Phone (925) 249 3000

Fax (925) 426 2556

info@cswl.com

<http://www.cswl.com>



Digital Signatures and PKCS#11 Smart Cards

Concepts, Issues and some Programming Details

A Technical Report

INDEX

INTRODUCTION..... 3

PUBLIC KEY CRYPTOGRAPHY..... 3

DIGITAL SIGNATURES & DATA ENCRYPTION..... 5

SMART CARDS AND OTHER CRYPTOGRAPHIC TOKENS..... 6

BASICS OF PROGRAMMING WITH PKCS #11 7

 PKCS#11 OBJECT MODEL: 7

 KEY-PAIR GENERATION: 7

 KEY-GENERATION MECHANISMS: 8

 SIGNING PROCESS: 8

Initialize the token: 8

Compute hash: 9

Sign the Hash: 9

Retrieve Certificate: 10

Clean-up: 10

 VERIFYING THE SIGNATURE: 10

Initialize the token: 10

Compute Hash: 11

Verify the signature: 11

Validate Certificate chain: 11

Clean-up: 12

 VERIFICATION USING MICROSOFT CAPI : 12

Acquire CAPI Context: 12

Compute Hash: 13

Verify the Signature: 13

Validate Certificate chain: 13

Release CAPI Context: 13

 VALIDATING A CERTIFICATE: 13

ONLINE CERTIFICATE STATUS PROTOCOL (OCSP) 15

SOME APPLICATIONS OF DIGITAL SIGNATURES..... 16

SOME ISSUES - THERE’S MORE WORK YET 16

Introduction

With the need for information security in today's digital systems growing, cryptography has become one of its critical components. Digital signatures are one of the many uses of cryptography. PKCS#11, also known as Cryptoki, was defined by RSA and is a generic cryptographic token interface. This article deals with applying digital signatures on documents using cryptographic smart cards and readers. We shall also discuss using MS Crypto API for verifying these digital signatures.

It is assumed that the reader has some idea of cryptography. We will not be explaining the complete PKCS#11 standards, which is available as a part of PKCS documentation. However, some fundamentals will be explained. Readers not inclined to programming details may skip the middle portion of this document.



Public Key Cryptography

Public-key cryptography allows one to digitally sign and encrypt information transacted between parties. Public Key Infrastructure (PKI) uses this technology and adds authentication and non-repudiation of the information regarding the parties concerned. Public Key Cryptography Standards (PKCS) is a suite of protocols and algorithms that are used as an industry standard when implementing public-key cryptography and infrastructure. The fundamentals are based on Key Pairs, Message Digests and Certification. These are described below.

A key pair consists of a private key and a public key. The private key is never revealed to any party. The public key is made available to the world, or at least the parties concerned with receiving or sending information. In public key algorithms like those from RSA, any data encrypted with the private key can be decrypted only with the public key, and data encrypted with the public key can be decrypted only with the private key. Stronger encryption uses longer keys. For strong encryption, it

is “computationally infeasible” to derive the private key given the public key, or vice versa.

Message Digests are hash functions that take in data and generate a statistically unique digest, like a 20 byte number – such that even one bit change in the input data results in a totally different digest. Thus these digests serve as finger-prints of a document. Given a digest and a document, and knowing the hash algorithm, it is easy to verify whether the digest is derived from the document.

Certification is the mechanism by which authenticity is established. A party generates a key pair consisting of the private and public keys. The public key is placed into a certificate request and sent to a certifying authority (CA) like Thawte, IDCertify, VeriSign and so on. The certifying authority (CA) verifies the party’s credentials and the purpose of using the keys, through a vetting process, and then certifies the public key they received. That is, the authority issues a certificate, typically called an X.509 digital certificate that contains the details of the party, the intended use of the certificate and most importantly, the party’s public key. This information is then digitally signed by the CA using the CA’s private key. The authenticity of the certificate itself can be verified by using the CA’s public key, which is made available from the CA’s web site, or comes embedded in a browser by default.

In essence, if you trust the CA, then you can trust that the public key in the verified certificate indeed belongs to who ever the CA says it belongs, and therefore if a digital signature on a document is verified using that public key, the information therein was indeed signed by the party mentioned in the certificate. This establishes authenticity, since only the holder of the corresponding private key could have created that digital signature. And trust in the CA is at the core of this process. If a CA is granted a notary or equivalent status, then the certificate and the information signed or encrypted cannot be repudiated and is valid in many courts of law.



Digital Signatures & Data Encryption

A digital signature is a digital attestation of a document by a party. This is to establish authenticity. A digital signature is an encrypted digest (hash) of the data to be signed.

One essentially creates a digest or hash (using an algorithm like MD5 or SHA1) from the document data and then encrypts this hash with one's private key. The encrypted hash thus becomes a digitally signed finger-print for that document, called a digital signature. This signature can now optionally be attached to the document, along with one's certificate. Anyone intent on verifying the digital signature would verify the certificate for authenticity first, then take the public key from the certificate and then verify the digital signature. The latter part involves decrypting the digital signature with the public key to reveal the digest or hash value. The document is then hashed using the same algorithm to check whether the digest values match.

A digital signature is typically attached to a document. This can be difficult for certain document types. It is required to embed the signature into the document without changing the document (!), which is contradictory. Therefore a signing process only works on the information portion of a document, and uses other sections of the format to embed the signature. For example it is possible to embed signatures into a Word document treating the latter as an OLE compound document. One may also store signatures as attributes of such a document. PDF is another format that is amenable to embedding using the DIGSIG API. Another technique is to create a container document (having a different naming extension) that includes the source document and the signature, and from which either can be extracted. Multiple signatures may be created and attached to a document. The signatures may be peer level or hierarchical level. Peer level signatures imply that one or more parties have endorsed the document by applying their signatures. Hierarchical signatures imply a work-flow and counter-signing process.

Creation of a digital signature involves using one's private key. In contrast, encryption of information meant for another party uses the other party's public key. Anyone, knowing that party's public key can send encrypted information. Only that party can decrypt the information, using his/her own private key.



Smart Cards and other Cryptographic Tokens

Smart cards are small, tamper-resistant devices providing users with convenient storage and processing capability. The smart cards are suitable for cryptographic implementations because they contain many security features that enable the protection of sensitive cryptographic data and provide for a secure processing environment. The protection of the private key is critical for digital signatures. This key must never be revealed. The smart card protects the private key, and many consider the smart card an ideal cryptographic token. The private key is generated by the smart card and never leaves the smart card providing high level of security.

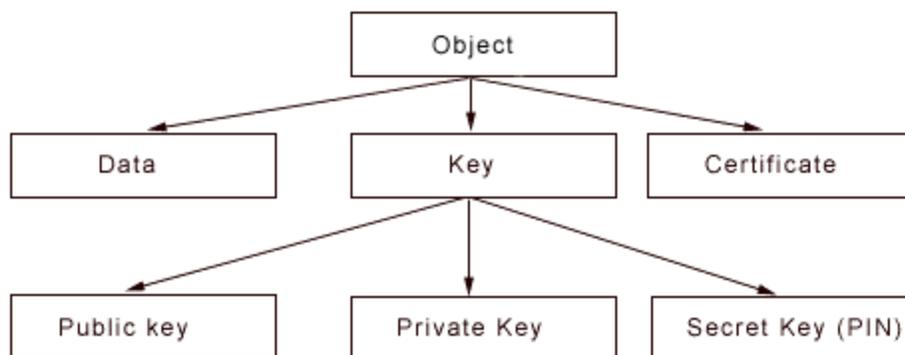
The PKCS #11 standard describes a programming interface that can work with cryptographic tokens and devices such as smart cards and PCMCIA cards. Through this interface one may initialize such devices, create key pairs, store certificates, digitally sign data etc. A PKCS#11 interface is typically implemented by a software driver that works in conjunction with the hardware reader and the token, and translates interface calls to token-specific controls. This well-defined interface allows even a browser to interact with such a token and make use of its capabilities.



Basics of programming with PKCS #11

In the following sections we will look at the PKCS#11 object model, see how to sign a chunk of data (for example a word document) and later verify the integrity and authenticity of the data. We shall also see the process involved in verifying the data using CAPI (Microsoft Crypto APIs) so that a smart card is not required for verification process. Due to the limited scope of this article we will not examine the process in great details.

PKCS#11 Object Model:



As seen in the diagram, this is a simple object model but powerful enough for the operations expected from a token. The private key is used for generating a digital signature and the public key (which is also a part of the certificate) is used for verification of the digital signature. Secret key or PIN is used for managing keys and certificates from the store. A token can typically store a number of such objects.

Key-Pair Generation:

Private key – public key pair is the core of cryptography. As mentioned earlier, the private key never leaves the crypto token. This mandates that generation of key-pair be carried out on the token itself. PKCS#11 also provides functions for key-pair generation on the token. C_GenerateKeyPair function is used for key-pair generation on the token. This function is used as follows.

```
Status C_GenerateKeyPair(hSession,pMechanism,pPublicKeyTemplate,  
usPublicKeyAttributeCount,pPrivateKeyTemplate,usPrivateKeyAttributeCount,  
phPrivateKey,phPublicKey);
```

Note that the data returned for private key is a handle to private key and not the private key. Another parameter of importance is the mechanism. Widely used mechanisms are DSA and RSA. These are explained in the next section.

Key-Generation Mechanisms:

Digital signature Algorithm and RSA (developed by RSA labs) are most widely used key-pair generation mechanism used in the industry. The major difference between these two mechanisms is that RSA keys are stronger and verification process is faster if RSA mechanism is used. DSA mechanism is faster during the signing process.

Signing Process:

The whole signing process can be broadly divided in to following steps.

1. Initialize the token
2. Compute hash (digest) of the data to be signed
3. Sign the hash using private key
4. Retrieve the certificate corresponding to the private key used for signing
5. Clean-up

Initialize the token:

To start the signing process we need to initialize the token. This is done in order to determine the number of cards and key-pairs present on the token. As a secret PIN protects the access to private key on the token, the PIN needs to be supplied to the token. This process also starts a new session on the token. It is essentially a call to C_Initialize function, followed by a C_Login() call.

Compute hash:

After initializing process we need to calculate the digest (hash) of the data to be signed. Hashing using PKCS#11 functions is a 3-step process. First we call C_DigestInit , then C_DigestUpdate function is called repeatedly to feed the whole document in chunks. Finally we call C_DigestFinal to flag the end of document data.

Extract of the code would look like

```
status = C_DigestInit(currentSession, &mechanism );  
status = C_DigestUpdate(currentSession, buf, bufSize);  
status = C_DigestFinal(currentSession, hash, &hashLength);
```

Note that in order to emphasize on the crypto functions, error-handling code is removed from the code fragment.

Sign the Hash:

This step is responsible for generating the signature for the hash computed by the previous step. As the private key never leaves the PKCS token, the data to be signed (hash in our case) needs to be passed on to the PKCS token. This is a 2-step process. We first call C_SignInit, function and then make a call to the C_Sign function, which results in a signature being generated. The signing method is illustrated below.

```
status = C_SignInit( currentSession, &mechanism, objPrivateKey);  
status = C_Sign(CurrentSession,hash,hashlen,pSignature, &uxSignatureLen);  
Mechanism could be set either to RSA or DSA.
```



Retrieve Certificate:

Once we have the signature, we need to retrieve the corresponding certificate, so that it could be sent along with signature for verification. Retrieving certificate from a token is simple and we need to call C_FindObjects, which can be used for finding any object on a token. The code essentially looks like

```
status = C_FindObjectsInit(currentSession, template1, size1);  
status = C_FindObjects(currentSession, objectPublicKey, objectCount,  
&objectCertificateCountFound);
```

Clean-up:

Finally we need to close the current session to free up any resources being used by the system. Apart from freeing up any object created by us we need to call C_CloseSession.

Verifying the signature:

Verification of the signature using public key through PKCS#11 compliant token is carried out in following steps.

1. Initialize the token
2. Compute hash (digest) of the data to be verified
3. Verify the hash using public key
4. Validate the certificate chain.
5. Clean-up

Initialize the token:

First we need to initialize the token for verification. This is done as shown below.

```
status = C_VerifyInit( sessionHandle, &mechanism, hObjectPublicKey);
```

Here we pass the public key, which is used for verifying the signature. The session handle is a valid open session handle.

Compute Hash:

As during signing process only the hash of data was signed, if we try to verify the document content instead of its hash, the verification process would fail. Hence we need to compute hash in the same manner we did for signing. Hashing using PKCS#11 functions is a 3-step process. First we call C_DigestInit, then C_DigestUpdate function is called repeatedly to feed the whole document in chunks. Finally we call C_DigestFinal to flag the end of document data.

Extract of the code would look like

```
status = C_DigestInit(currentSession, &mechanism );  
status = C_DigestUpdate(currentSession, buf, bufSize);  
status = C_DigestFinal(currentSession, hash, &hashLength);
```

Verify the signature:

Once we have the hash we can use C_Verify to validate the data using the public key supplied during initialization step.

```
status = C_Verify( sessionHandle, pbHash, uxHashLen, pSignature,  
uxSignatureLen);
```

This function returns true if the verification was successful else it returns false.

Validate Certificate chain:

This is explained in 'Validating the certificate' section below.



Clean-up:

Finally we need to free up all memory and objects allocated by us.

Verification using Microsoft CAPI :

Most of the time, recipient of signed document does not have access to the PKCS#11 compliant token for carrying out the verification process described in previous section. In such cases verification could also be carried out using Microsoft Crypto API. This functionality is available on all machines with Internet Explorer installed on them. This way we can carry out the verification process without a token. One important point to remember while verifying the signature using CAPI is that the signature Byte sequence expected by CAPI is reverse of the byte sequence generated by a PKCS#11 compliant token. If the byte sequence is not reversed before verification then CAPI would never be able to verify our signatures.

Steps involved in verification using CAPI (apart from reversing signature byte sequence) are

1. Acquire CAPI Context
2. Compute Hash
3. Verify Signature
4. Validate certificate chain
5. Release CAPI Context

Acquire CAPI Context:

First of all we need to get handle to the default provider. This is done as under

```
CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0))
```

Compute Hash:

Hashing using CAPI is done in 2 steps, first we create the hasing object using CryptcreateHash, then we supply chunks of data in a loop using CryptHashData.

```
CryptCreateHash(hProv, &mechanism, 0, 0, &hHash))
```

```
CryptHashData(hHash, pbBuffer, dwCount, 0)
```

Verify the Signature:

As we had signed the hash, we need to supply the computed hash to the CryptVerifySignature function. It will return true if the signature is valid else return false.

```
CryptVerifySignature(hHash, pbSignature, dwSignatureLen, hSigPublicKey,  
szDescription, 0)
```

Validate Certificate chain:

This is explained in detail in ‘Validating the certificate’ section below.

Release CAPI Context:

Finally we need to release the acquired CAPI context using

```
CryptReleaseContext(hProv, 0);
```

Validating a Certificate:

Verification step only verifies the integrity of the document, it only tells us that the document was signed by a particular key-pair and it has not been modified after signing. It is always advisable to first verify the certificate before verifying the data itself. This helps us in determining whether the certificate is authentic or was tampered with.



Important thing to remember about verification step is that It does not validate the identity of the signer. Authenticating the signer information is critical to prevent anyone generating key-pairs with some one else's identity. This is ensured by a Certificate Authority (CA) who carries out reasonable identity checks before issuing a certificate to a person or company. The CA will have its own key pair and its own certificate, which is self signed, also known as a root certificate. The CA's private key is used to sign all issued certificates. Sometimes a CA will be a 'channel CA' to some other certifying authority and thus a subsidiary. In such a case the signatures can be traced up along certificate chain to a root certificate authority. The root certificate is self-signed. Unless the entire certificate chain is validated, a given certificate and a given signature cannot be guaranteed to be valid.

To validate a certificate, one needs to validate the digital signature on it. This requires parsing the X.509 certificate and identifying the content and the digital signature. Parsing can be accomplished by using an ASN.1 parser (3rd party or hand coded) against the ASN.1 description of an X.509 certificate. There are also libraries available that parse X.509 certificates. Microsoft CAPI and PKCS libraries are examples. Once the data content and the digital signature are located, the data must be hashed to obtain a digest. Next, the certificate authority's root CA certificate must be parsed to extract the public key. This certificate is typically placed in a trusted root certificates repository. The public key is then used to decrypt the digital signature on the certificate and reveal the hash. This hash must match with the computed hash on the data.

If the issuing CA's certificate is not marked as 'trusted' in the certificate repository, one must go further up the chain by identifying the next higher level CA. This can be done by parsing and inspecting the issuing CA's root certificate. The digital signature on the issuing CA's root certificate now must be verified using the root CA certificate of the higher level CA. And if this CA happens to be not trusted, go another level up. When one finds a root CA certificate that is marked as trusted, the

validation process stops successfully. To hand code this functionality, one can use the certificate repository functions, like the CertXXX() functions in CAPI.

It is not sufficient to trust a certificate because the issuing authority is trusted. The authority may have revoked the certificate. One of the ways to validate the certificate chain for revoked certificates (certificates which are not expired, but revoked by the issuing CA), is through Certificate Revocation Lists (CRLs). A certificate revocation list (CRL) is a digitally-signed, time-stamped black-list of revoked but unexpired certificates, issued by a CA periodically, They have limitations in that the CRL on hand may not be considered sufficiently fresh, and they may grow to an unacceptably large size. The other more efficient way of validating the certificate chain is using Online Certificate Status Protocol (OCSP)



Online Certificate Status Protocol (OCSP)

Some applications—such as high value funds transfer—require immediate on-line checking of a certificate's status, rather than tolerate any latency as is inherent in all CRL-based mechanisms. OCSP specifies a transaction whereby a certificate-using application can obtain from a CA a 'digitally-signed' indication of the current status of any certificate. While OCSP has excellent timeliness characteristics, it may present performance problems in comparison with CRLs, and is therefore not suitable for all applications.

OCSP protocol is defined in the Abstract syntax one (ASN.1) notation and has a client and server part associated with it. Any client that needs to verify a certificate's current status can invoke this protocol, connect to the certificate authority's server over a secure channel like HTTPS and send the certificate serial number and hash value to the server. The server returns the status of the certificate whether it is valid, revoked, unknown state etc. This information is signed by the certifying authority, so it may be stored away for later use. Of course, there may be nominal fees involved for this transaction.



Some applications of Digital signatures

Digitally signed documents provide the authenticity of paper documents, and convenience of electronic documents. This has made digitally signed documents a great success. Some of the applications of digitally signed documents are in “legal document” centric industries like law firms, banking, stock broking. It also has high potential in government sector, where powerful document management software can make the system more efficient and fast.

Digital signatures also make transactions happen much faster. For example the ownership of a cargo carried by an oil tanker from the Persian Gulf to New York may change hands 5 to 10 times by the time the tanker reaches New York. With digital signatures applied to contracts on e-Marketplaces, they may change hands maybe 50 times or more, greatly speeding up the trading process.



Some issues - There's more work yet

Even though PKI is established and cryptographic standards are in place, not much of standardization has occurred on how digital signatures are actually used in a software application context. We attempt to highlight some deficiencies below:

- As of this writing, not many document formats have official digital signature embedding capabilities.
- For some that do, proprietary methods are used so that signing and verification software from one provider cannot work with that from another, simply because there is no agreement on where and how to store these signatures in a document.
- There is little by way of concept on hierarchical and peer level signatures. We believe this terminology has not been used before in the industry. The

semantics of applying multiple signatures need description and standardization.

- OCSP as it is defined today is not a scalable solution
- There is little by way of a digital signature repository defined. One probably needs CA, service provider or enterprise level repositories where OCSP-type responses and digital signatures could be stored for archival and look up purposes.
- Currently to establish authenticity of a digital signature, one has to trust both the CA's vetting process and the signing solution provider's proprietary techniques of embedding and verification. As of this writing we know of no code and functionality audit established whereby a software solution provider's solution can be certified. Nor is it easy to do until document formats support this feature natively. In which case, the responsibility falls in addition to the owner of the document format as well as the owner of the software product that generates and manipulates the document format. These can be hair-splitting issues for legal discussions.

Even though PKCS currently consists of mathematically verifiable procedures which can prove authenticity, many implementations with respect to digital signatures today have gaping holes mentioned above which make them practically ad hoc and unverified. Further discussion on this issue is beyond the scope of this article.



Copyright Notice:

2002 California Software Labs. All rights Reserved. The contents on the document are not to be reproduced or duplicated in any form or kind, either in part or full, without the written permission of California Software labs. Product and company names mentioned here in are the trademarks of their respective companies.