

Tutorial: Windows Cryptographic API, A Session Keys Paradigm

Dr. Edwin A. Hernandez,
MOTOROLA Inc,
JAVA iDEN group
8000 West Sunrise Boulevard
Plantation, FL, 33322

Introduction

Many protocols rely on session keys to diminish the possibilities of eavesdropping and preserve the secrecy of private keys. As an example, Diffie-Hellman [1] uses two prime numbers, called p and g to generate a master key, via an exchange of two random private numbers a and b . Public values are derived from $g^a \bmod p$ and $g^b \bmod p$. The values of p and g are large prime numbers and even though, a might see the exchanged values; it's not possible to determine the master key, k , from the values exchanged and for this specific session. In general, private keys are stored as part of the code in plaintext, or obfuscated within the code. Key storage in this format is not structured and, in many situations, is unable to protect the secrecy of private keys from unauthorized users. This tutorial shows how session keys are used in Microsoft's Crypto API to protect user's keys.

Session keys in Crypto API

Microsoft's Crypto API (MSCAPI) [2] uses session keys as the norm not the exception. The real key value used for the private key is preserved and protected by the Cryptographic Service Provider (CSP). Windows CE MS CAPI has a couple of CSPs: RSA Base, RSA Enhanced, as well as many others. Additionally, new CSPs could be created and incorporated into Windows and Windows drips CE (Figure 1).

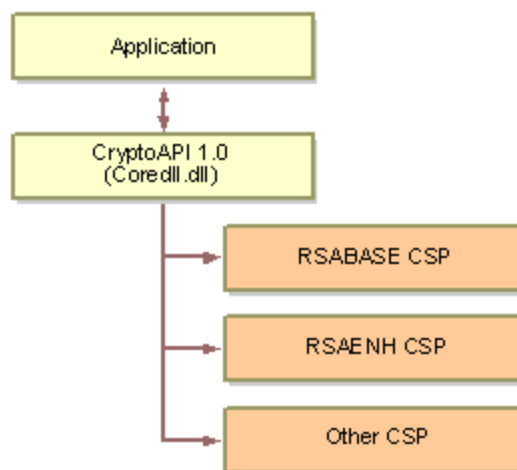


Figure1. Cryptographic Service Provides in Windows CE [2]

These CSPs keep the keys and all the containers for those keys isolated from the application layer. By opening a container, application are then able to sign, encrypt, and decrypt data with a key. The container could be password protected, stored in a smartcard, or a secure media. Two methods can be used to set keys in a CSP.

1. CSP Generated key (e.g. via CryptGenKey or CryptDeriveKey)
2. Force a session and application keys be the same.

In the first option, key generation is rather simple:

- 1) First, we create a container (e.g. “Keytest1” in Figure 2) and if this value is set to NULL then a default container are used. This concept is also used in the desktop as shown on Figure 2 (e.g. signcode.exe). In a Windows machine, during the signing process and a private key you can select the container you want to use. You could protect these containers with a password or store them in a smartcard; these containers are not preserved in Windows CE and should be manually stored in the registry.

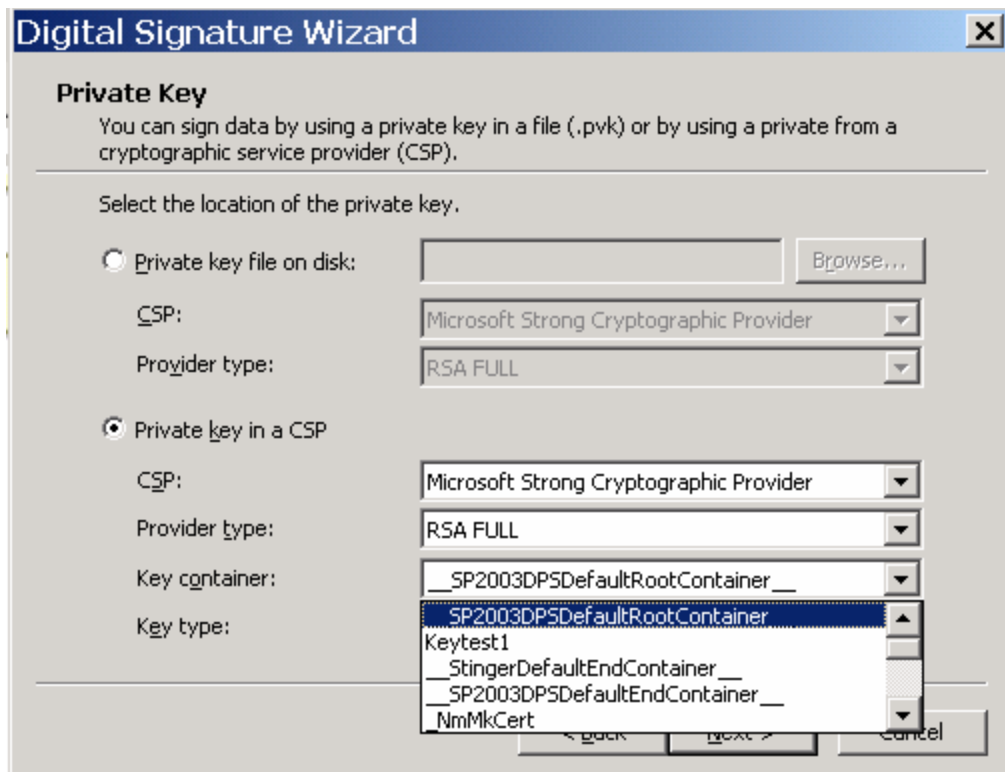


Figure 2. Selecting your container where a Private key is stored.

- 2) The CryptGenKey call is used to create a new key value. This key could be DES, 3DES, RSA Public/Private key, etc. All containers created could have a separate a signature and exchange key pairs (AT_SIGNATURE, AT_EXCHANGE) (Figure 3)

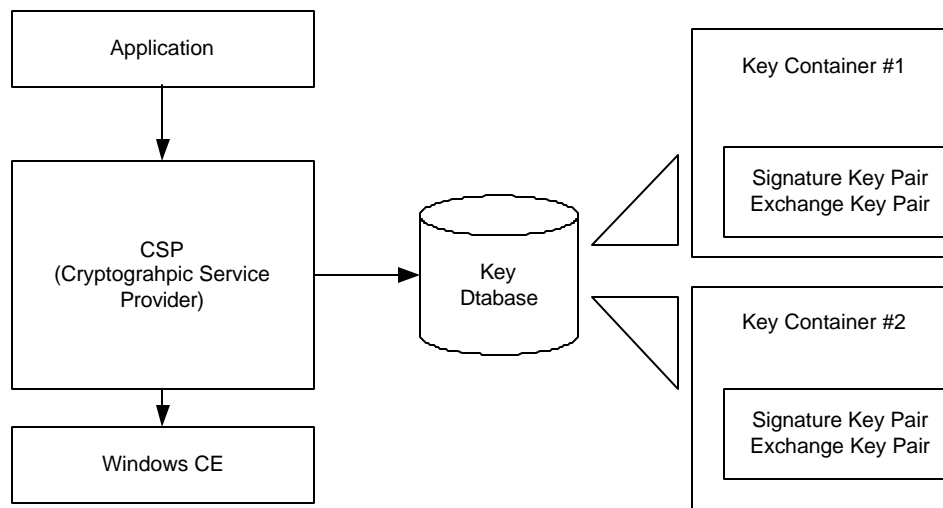


Figure 3. Special containers for AT_SIGNATURE or AT_EXCHANGE [2]

The application has access to a session key. A session key might be signed and therefore protected by the CSP's internal Public and Private Key pair.

Assuming, you created a 3-DES key that you stored in the registry. If a malicious user reads the key and its signature. The key is useless without the proper Private Key that the container used to export the key to the application space.

- 3) The container name could be preserved for future used or destroyed. CryptDestroyKey() destroys the key and the container will be empty.

How do we access the containers and the session keys, once generated? MS CAPI provides CryptImportKey(...) and CryptExportKey(...) calls are used to import and export: Private Key blobs (e.g. RSA private key), Simple Key Blobs (e.g. 3-DES), and Public Key blobs (e.g. RSA public key blobs).

Session keys could also be kept and preserved for future use (Taken from [4]):

- Create a simple key BLOB using the [CryptExportKey](#) function. This transfers the session key from the CSP to your application memory space. Specify that your own key exchange public key be used to encrypt the key BLOB.
- Store the signed key BLOB.
- Read the key BLOB from storage when you need to use the key.
- Import the key BLOB into the CSP, using the [CryptImportKey](#) function.

This is a common situation to share a key between two local processes. However, when a network or two or more machines are involved, sharing is more complicated. Why?

Two CSPs in two different machines are totally different, the same container will have two different AT_SIGNATURE and AT_EXCHANGE key pairs.

Figure 4 shows the method suggested by Microsoft when exchanging session keys. Since the session key needs can be imported into a MS CSP (Microsoft's Desktops or Windows CE machines), the session key needs to be encrypted by the receiver's Public Key and later on decrypted by the receiver's private key. The session key could also be exchanged using a separate SSL channel. The encrypted message is sent out to the requester encrypted with the session key.

The "distributed case", in Figure 4, also represents the events taking place at the local case. Internally, the CSP uses its own public key to export the key blob and the session key used cannot be imported in to the CSP without knowing the Private Key from the CSP's endpoint. In the local case, source and destination are simply the same.

Warning: *The container's AT_SIGNATURE and AT_EXCHANGE should not be exported without being encrypted. A container will be fully vulnerable to attacks and new key values MUST be generated.*

In pseudo code, this is what takes place when a message is exchanged:

Sender	Receiver
<pre> hKey = CryptGenKey(3DES) CryptImport(hKeyExchange, &publicKey); CryptExport(hKey, &SessionKey) CryptEncrypt(hKeyExchange, ..&SessionKey,) CryptEncrypt(hKey, &Message); Send(Message, SessionKey) </pre>	<pre> Receive(Message, eSessionKey) CryptImport(hKey, &dSessionKey); CryptDecrypt(hKey, &Message); </pre>

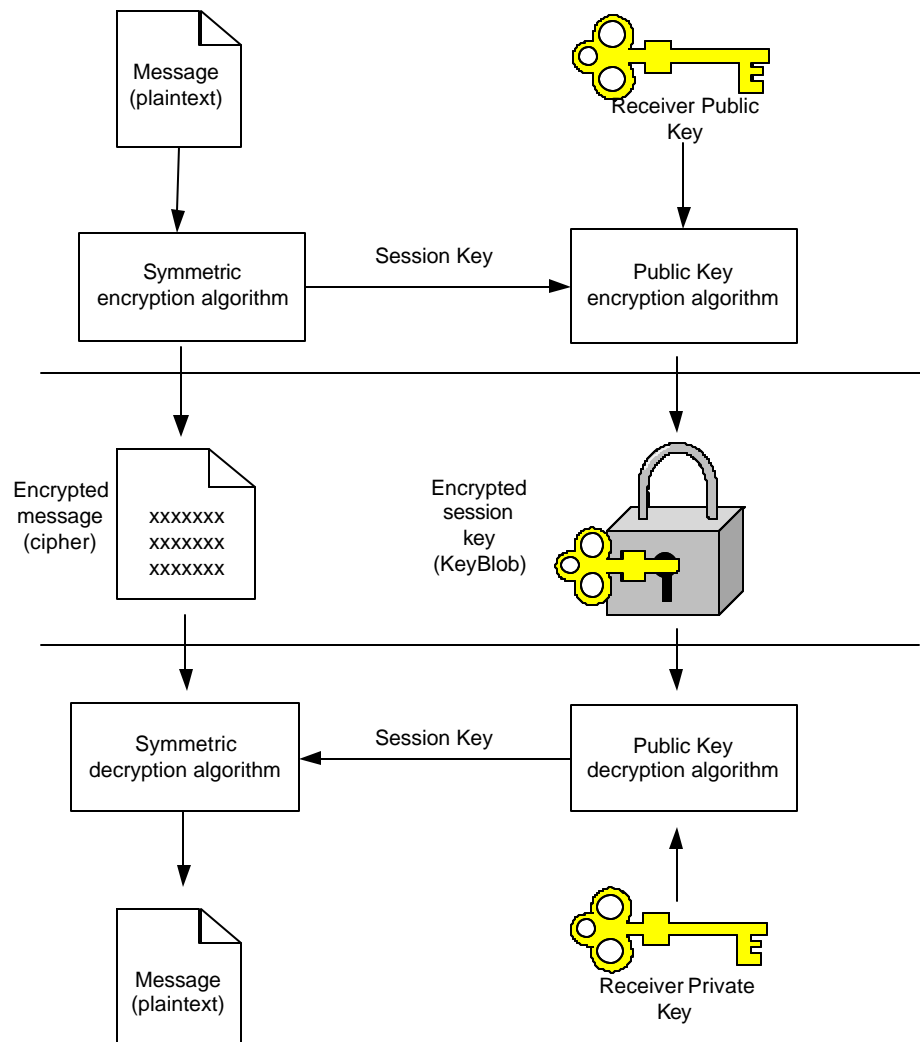


Figure 4. Exchanging session keys [2]

What if our receiver system uses Certicom or any other Crypto API? It is not possible to use the session keys into certicom because the session key was derived differently (using an unknown public/private key pair). This applies also to private keys stored in ROM or a smartcard.

We should then Import the keys from the application space into the CSP, or force the CSP's container to a certain key value. In order to import and/or export keys, and create the matching between session keys and container's keys, you have to do the following:

- 1) Generate a Key pair (CryptGenKey)
- 2) Export the session key from the key pair created (CryptExportKey).
- 3) Since, this is a Private Key Blob, the key should be modified by applying an exponent of one. A private key blob is represented by:

```
PUBLICKEYSTRUC publickeystruc ;
RSAPUBKEY rsapubkey;
BYTE modulus[rsapubkey.bitlen/8];
```

```

BYTE prime1[rsapubkey.bitlen/16];
BYTE prime2[rsapubkey.bitlen/16];
BYTE exponent1[rsapubkey.bitlen/16];
BYTE exponent2[rsapubkey.bitlen/16];
BYTE coefficient[rsapubkey.bitlen/16];
BYTE privateExponent[rsapubkey.bitlen/8];

```

This representation of a private key is the Chinese remainder theorem and could be referenced in [5]. According to the RSA standard, the encryption takes place by using:

- If the message representative m is not between 0 and $n - 1$, output “message representative out of range” and stop.
- Let $c = m^e \bmod n$.
- Output c .

If the exponent, $e=1$ no transformation will occur when importing the key into the CSP. This key is used to import and export the Application and CSP keys [6].

- 4) Once the new key is set, the key is imported into the CSP by `CryptImportKey(...)`. The `CryptImportKey` API allows the use of two HKEY values, the key itself being imported, and a key used to make certain transformation (`hPubKey`, in Crypto API reference).
- 5) Any subsequent `CryptImportKey` calls can then use the key that does nothing to the key that’s being imported and both session and CSP keys match.

In summary, the application developer has to choose the use of session keys managed by the CSP or match session keys to CSP keys. Therefore session keys could then be fully managed by the application, handled by the CSP, or a combination of both approaches. *If direct mapping of session and CSP keys is used, exposure is inevitable and it’s not recommended. Your secret is not protected!* The use of session keys is safer because the CSP will use its own key pair to protect the real key and only the session key is available to the applications. Application developer’s could then later on update the keys and avoid a persistent session key to remain valid for a long period of time.

Links and References

- [1] RSA Security FAQ: <http://www.rsasecurity.com/rsalabs/faq/3-6-1.html>
- [2] MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcesecur/html/_wcesdk_Cryptography.asp
- [3] MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcesecur/html/_wcesdk_exchanging_cryptographic_keys.asp
- [4] MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcesecur/html/_wcesdk_exchanging_cryptographic_keys.asp
- [5] PKCS #1: <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.doc>
- [6] RSA Exponent of one (transformation):
<http://support.microsoft.com/default.aspx?scid=http://support.microsoft.com:80/support/kb/articles/Q228/7/86.ASP&NoWebContent=1>
- [7] Mobile Devices: <http://www.windowsfordevices.com/articles/AT6765599475.html>