BLACK HAT DC 2009

Windows Vista Security Internals

Michael Muckin
Lockheed Martin
Corporate Information Security Engineering

2/18/2009

An analysis of Vista vs. prior versions of Windows' implementation of cryptographic primitives to protect sensitive in-memory data, such as password hashes. Vista SP1 introduces significant changes to these mechanisms and will change the way analysts, researchers and attackers approach the acquisition of protected in-memory data. A basic overview of pre- and post-Vista security architecture is given to provide some groundwork for the detailed analysis.

Contents

SECTION I

Overview of the Changes

Introduction

Windows Vista introduces some significant changes in its security architecture; there are also some major changes between Vista RTM (release-to-manufacturing; a Microsoft term for "gold" code or SP0) and Vista SP1. This paper will first provide a brief overview of some of those architectural changes, and will then focus on the specific components that will be covered in greater detail in subsequent sections. The focus areas for this paper will be on logon and authentication, encryption and some networking enhancements in IPSec.

Most of the discussion in this paper will focus on the Local Security Authority service - LSASRV.dl, the new Cryptography Next Generation (CNG) APIs – BCrypt.dll and NCrypt.dll, and some enhancements to IPSec contained in IKEEXT.dll.

Overview of the Vista Security Architecture

To lay some basic groundwork, let's first review some of the major changes in the security architecture of Windows Vista vs. prior Windows operating systems.

Windows Vista has produced many changes that are security feature/function related and some that are low-level changes that help combat malicious software from executing. Things like Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), User Account Control (UAC) are all intended to make Vista a more secure system – these technologies have been adequately discussed elsewhere and we will not dive deeply into them here.

There have also been some specific architectural changes to Vista that focus on the discrete security modules, functions, APIs and components that handle logons, encryption and networking. We will first discuss these changes at a high level and then dive into specific details in later sections.

Pre-Vista Security Architecture

For comparison, we will first review some of the major components of the Windows security architecture pre-Vista. The diagram in Figure 1 illustrates the architecture for logon and authentication, which includes the following components:

- LSASS – Local Security Authority Subsystem
  - LSA interfaces with many other security related components, all of which are not listed here. For our purposes, LSASRV.dll is the most important component.
- WinLogon – The user-mode process that provides interactive logon services
- MsGina – The user interface for WinLogon
- Authentication Packages – Authentication packages validate a user's logon credentials. In Windows there are three primary packages:
  - MSV1_0 – authentication package for LM/NTLM authentication requests
  - Kerberos – for Windows Domains
  - SPNEGO – a package to select between Kerberos (preferred) or NTLM

  It is possible to create custom authentication packages that can be implemented as a DLL.

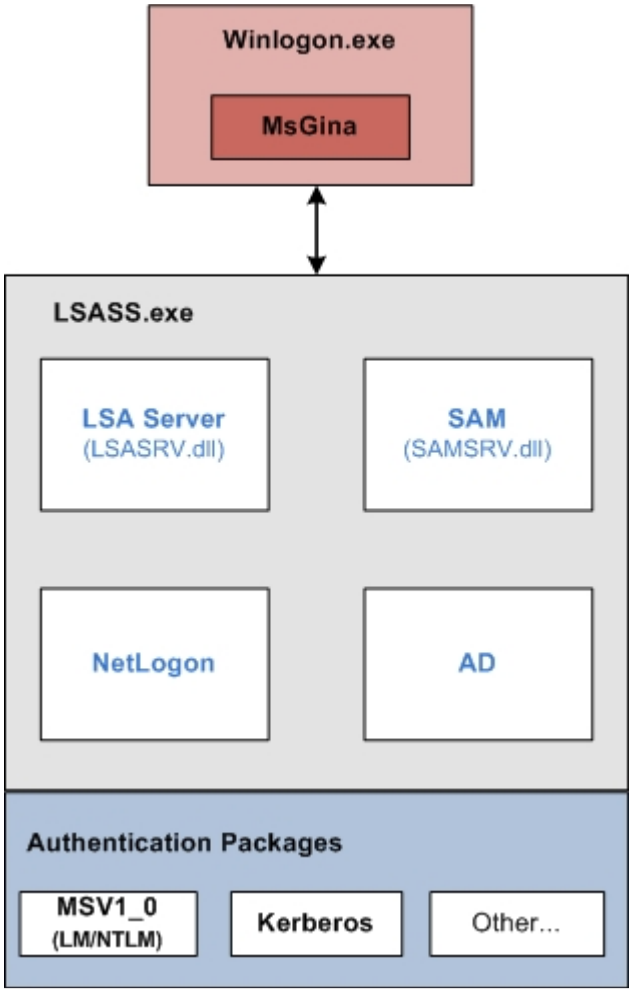


Figure 1 - Pre-Vista Security Architecture

Figure 2 provides details about the Windows Logon architecture. This is a big topic – a whole paper on just the Windows logon process could be (and has been) written; we will focus on the important concepts relevant to this paper. Session 0 contained all services, including system services and the interactive logons of any users on the system.

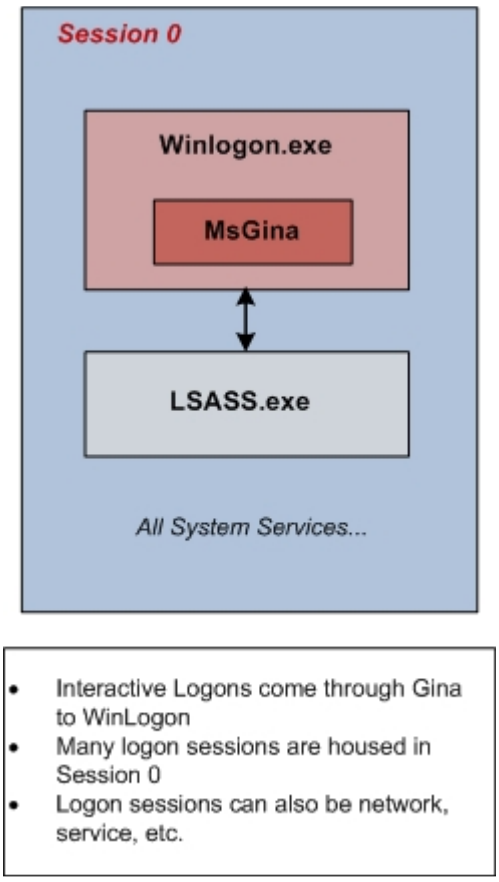


Figure 2 - Pre-Vista Logon Architecture

Also, unless it was replaced by a third-party, all interactive logon requests were processed by MSGina (msgina.dll), which provides the GUI to capture user's input to pass to WinLogon, which would then pass on to LSASS and the appropriate Authentication Package.

Vista Security Architecture

The Vista security architecture has departed from two major components that were discussed previously – namely MSGina and the implementation of Session 0. We will also illustrate some modifications in how Vista implements the Windows Integrity Mechanism/Levels to provide separation of privileges and processes. Lastly, there is the upgrade to the Crypto API – which is called the Cryptography Next Generation (CNG) API. It is important to understand these changes since they will be a large part of the remainder of this paper.

Logon Architecture

Vista has a redesigned architecture for processing both interactive (i.e. logging in from the keyboard) and other logons, such as service and network logons. Session 0 now contains only System Services and does not process interactive logons. I have found one discrepancy to this, however, in the `__vmware_user__` user account (shown below in Figure 3). This account will show up as an interactive logon in Session 0 on Vista (I have not yet researched the details of this). All other true interactive logons seem to receive sequential logon sessions, i.e. Session 1, Session 2, etc. Other specific details of the new logon architecture are listed below and illustrated in Figure 4.

```
[7] Logon session 00000000:00049371:
    User name:    Vajra\__vmware_user__
    Auth package: NTLM
    Logon type:   Interactive
    Session:      0
    Sid:          S-1-5-21-1663435799-3885150212-2807305598-1014
    Logon time:   2/4/2009 6:49:59 PM
    Logon server: VAJRA
    DNS Domain:
    UPN:
```

Figure 3 - Interactive Logon in Session 0

Vista Logon Architecture changes:

- Removal of MsGina – it is probably more accurate to say that any Gina dll – whether it be Microsoft's or a third party, will not be utilized by WinLogon.
- WinLogon and Logon Sessions vs. Session0 – Separation of logon sessions for users and services. Concept is that services should have no need to interact with the desktop
- LogonUI (new) and Credential Providers (new) components – essentially the replacement for Gina. Credential Providers architecture is meant to be extensible and allow for multiple plug-in providers. WinLogon handles interactive logons directly. LogonUI is COM based

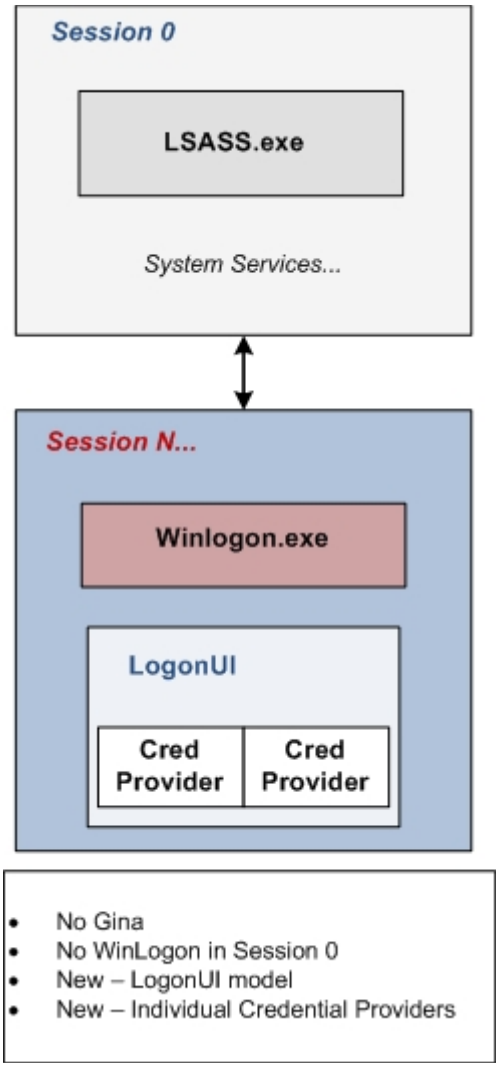


Figure 4 - Vista Logon Architecture

Integrity Mechanism

Another important component and enhancement of the Vista security architecture is the Integrity Mechanism. Although the Integrity Mechanism and Levels are not new in Vista, there are some changes in Vista vs. prior operating systems that are important for our purposes. Integrity Levels are one way that Windows enforces Mandatory Access Controls to specific system objects and resources. It also provides some level of separation of privileges for processes running in the same user's context. One goal of integrity levels is to isolate system services – especially those running in Session 0 – from all other processes.

The Integrity Mechanism adds an integrity level to security access tokens and a mandatory label for Access Control Entries (ACE) on the System ACL (SACL) of securable objects. There are five defined integrity levels:

Table 1 - Integrity Levels

Value	Integrity Level	Symbolic Name
0x0000	Untrusted	SECURITY_MANDATORY_UNTRUSTED_RID
0x1000	Low Level	SECURITY_MANDATORY_LOW_RID
0x2000	Medium Level	SECURITY_MANDATORY_MEDIUM_RID
0x3000	High Level	SECURITY_MANDATORY_HIGH_RID
0x4000	System Level	SECURITY_MANDATORY_SYSTEM_RID

These integrity levels are used to construct integrity level SIDs (Security Identifiers) so that they can be easily integrated into the existing Windows access control architecture. The unique SID value for integrity levels is S-1-16-XXXX. So, an integrity level SID for a High Level entry would be S-1-16-12288 (12288 is decimal for 0x3000).

There are also Mandatory Access Token Policies and Mandatory Label Policies specific to Vista that we should discuss here. They are listed in the two tables below.

Table 2 - Mandatory Access Token Policy

Policy	Description
TOKEN_MANDATORY_NO_WRITE_UP	Default policy on all access tokens; restricts write access on objects with a higher level than current token
TOKEN_MANDATORY_NEW_PROCESS_MIN	Controls behavior of child process integrity level assignment; Instead of inheriting parent level, an algorithm determines least possible privilege

Table 3 - Mandatory Label Policy

Policy	Description
SYSTEM_MANDATORY_POLICY_NO_WRITE_UP	Restricts write access by a subject with a lower integrity level
SYSTEM_MANDATORY_POLICY_NO_READ_UP	Restricts read access by a subject with a lower integrity level
SYSTEM_MANDATORY_POLICY_NO_EXECUTE_UP	Restricts execution by a subject with a lower integrity level

And to see how specific user account SIDs are mapped to Integrity Levels, see Table 4 below. Notice how all the system accounts (LocalSystem, LocalService, NetworkService) have the highest – System – integrity levels, and that Administrators do not have the highest levels by default.

Table 4 - Account SIDs and Integrity Levels

SID in access token	Assigned integrity level
LocalSystem	System
LocalService	System
NetworkService	System
Administrators	High
Cryptographic Operators	High
Authenticated Users	Medium
Everyone (World)	Low
Anonymous	Untrusted

The Integrity Levels add yet another obstacle to attackers and malicious users/code attempting to compromise a Vista machine. Some areas where it directly impacts a potential attacker is injecting code into system services and privilege escalation. Coupled with other protection mechanisms such as ASLR, DEP, SafeSEH it is becoming more challenging and complex to exploit and execute malicious code on Vista.

Encryption Services

Windows Vista first offers the use of the Cryptography Next Generation (CNG) service and APIs, as provided by the modules BCrypt.dll, Ncrypt.dll and the kernel driver Ksecdd.sys (there are probably other modules involved). CNG is the successor to the CryptoAPI. CNG is well documented on MSDN and there is a CNG SDK available.

Although the new CNG services were available in Vista, it wasn't until SP1 that Microsoft implemented some of the CNG services directly into the security components that comprise user logons and storage of password hashes. We will explore this area in detail in other sections of this paper.

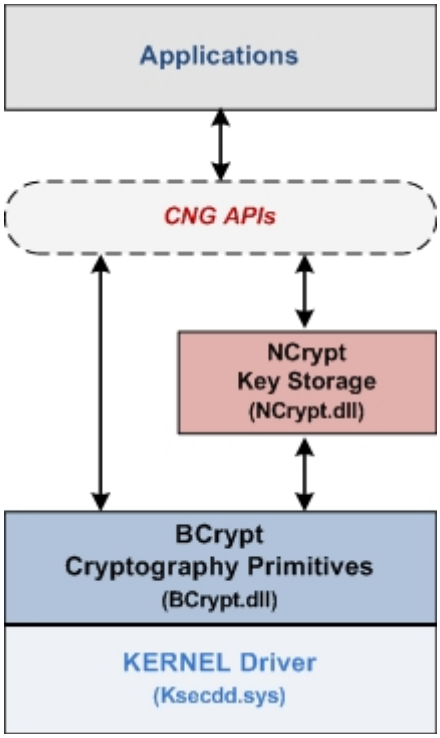


Figure 4 - CNG Architecture

This new CNG architecture is important to grasp as it has replaced the mechanisms used to protect password hashes stored in memory on Windows Vista SP1, and is the lion's share of discussion of this paper.

An interesting side note about CNG and SP1: the implementation of the random number generator in CNG as implemented in Vista SP1 (and Windows Server 2008) is not compliant with FIPS 140-2 RNG in several modules.

The author will speculate that this is so because of the addition of a number of "preferred" system functions used for RNG creation found in SP1 that are not present in RTM.

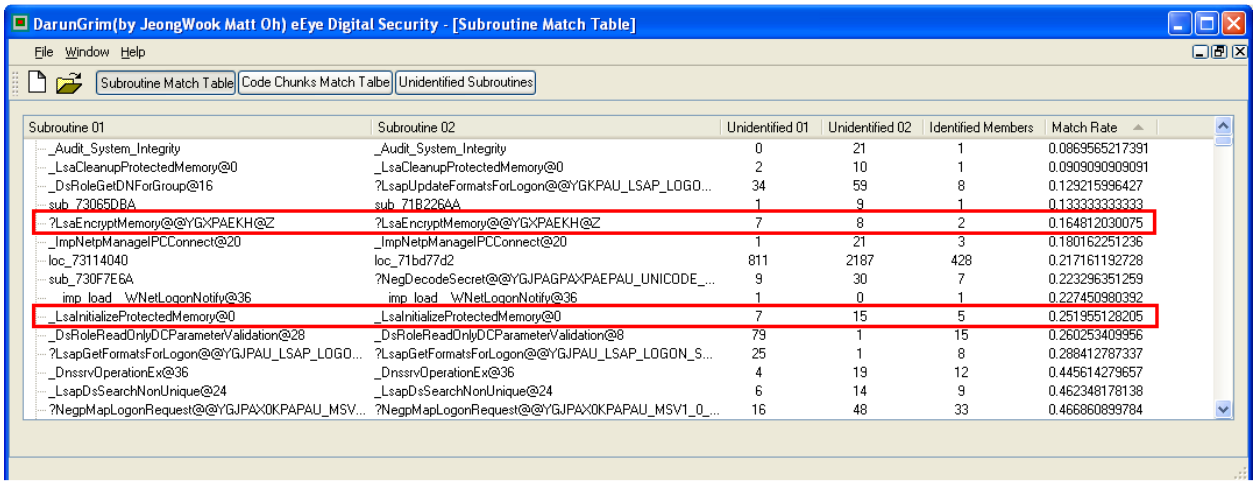
SECTION II

Protection of In Memory Password Hashes

A Change Is Detected

While doing some typical security testing as part of an analysis of malicious user capabilities on a compromised machine – and also to validate some countermeasures that were being developed to clear password hashes in memory – I was using the favorite technique of dumping the various password stores (hashes, credentials, LSA, etc.) on Windows machines. I performed these tests using the usual toolsets (PWDumpX, PTH Toolkit, etc.) on both XP and Vista computers with great success. I later attempted this same testing on another Vista machine and discovered that the dumping of hashes stored in memory was not possible, even after some significant tweaking and digging for the necessary offsets using PTH Toolkit. I soon realized the second Vista machine was at SP1 and the first Vista machine was at RTM. Further analysis revealed Vista RTM behaves exactly the same as XP SP3 when dealing with in memory stored hashes (at least as far as my testing could detect).

Disassembly and comparative binary analysis of the relevant modules and functions involved with the storage of in-memory hashes revealed that Vista SP1 has made some significant changes to these specific functions. The two functions most impacted are `LsaInitializeProtectedMemory()` and `LsaEncryptMemory()`, both contained in LSASRV.dll. LSASRV.dll provides the primary services of the Local Security Authority server that is offered via the Local Security Authority Sub-system (LSASS.exe) user-mode process. The native functions of LSASRV.dll provide services for authentication, encryption, user account privileges and management, domain trusts and more. It is a key component of the Windows security architecture.



Subroutine 01	Subroutine 02	Unidentified 01	Unidentified 02	Identified Members	Match Rate
_Audit_System_Integrity	_Audit_System_Integrity	0	21	1	0.0869565217391
_LsaCleanupProtectedMemory@0	_LsaCleanupProtectedMemory@0	2	10	1	0.0909090909091
_DsRoleGetDNForGroup@16	?LsapUpdateFormatsForLogon@@YGKPAU_LSAP_LOGO...	34	59	8	0.129215996427
sub_73065DBA	sub_71B226AA	1	9	1	0.133333333333
?LsaEncryptMemory@@YGXPAEKH@Z	?LsaEncryptMemory@@YGXPAEKH@Z	7	8	2	0.164812030075
_ImpNetpManageIPCConnect@20	_ImpNetpManageIPCConnect@20	1	21	3	0.180162251236
loc_73114040	loc_71bd77d2	811	2187	428	0.217161192728
sub_730F7E6A	?NegDecodeSecret@@YGJPAGPAXPAEPAU_UNICODE_...	9	30	7	0.223296351259
__imp_load__WNetLogonNotify@36	__imp_load__WNetLogonNotify@36	1	0	1	0.227450980392
_LsaInitializeProtectedMemory@0	_LsaInitializeProtectedMemory@0	7	15	5	0.251955128205
_DsRoleReadOnlyDCParameterValidation@28	_DsRoleReadOnlyDCParameterValidation@8	79	1	15	0.260253409956
?LsapGetFormatsForLogon@@YGJPAU_LSAP_LOGO...	?LsapGetFormatsForLogon@@YGJPAU_LSAP_LOGON_S...	25	1	8	0.288412787337
_DnssrvOperationEx@36	_DnssrvOperationEx@36	4	19	12	0.445614279657
_LsapDsSearchNonUnique@24	_LsapDsSearchNonUnique@24	6	14	9	0.462348178138
?NegpMapLogonRequest@@YGJPAX0KPAPAU_MSV...	?NegpMapLogonRequest@@YGJPAX0KPAPAU_MSV1_0_...	16	48	33	0.466860899784

Figure 5 - eEye Darun Grim Diffing Tool on LSASRV.dll

Using the Darun Grim diffing tool from eEye, it is obvious that there are significant differences in many functions contained within LSASRV.dll. By sorting on Match Rate within the tool (which calculates a percentage based on the differences detected between the identified functions) we see that LsaEncryptMemory and LsaInitializeProtectedMemory are two functions among the most dissimilar.

Then, using Darun Grim's diffing-graphing capability, you can see the side-by-side differences in execution of the same functions in Vista and Vista SP1.

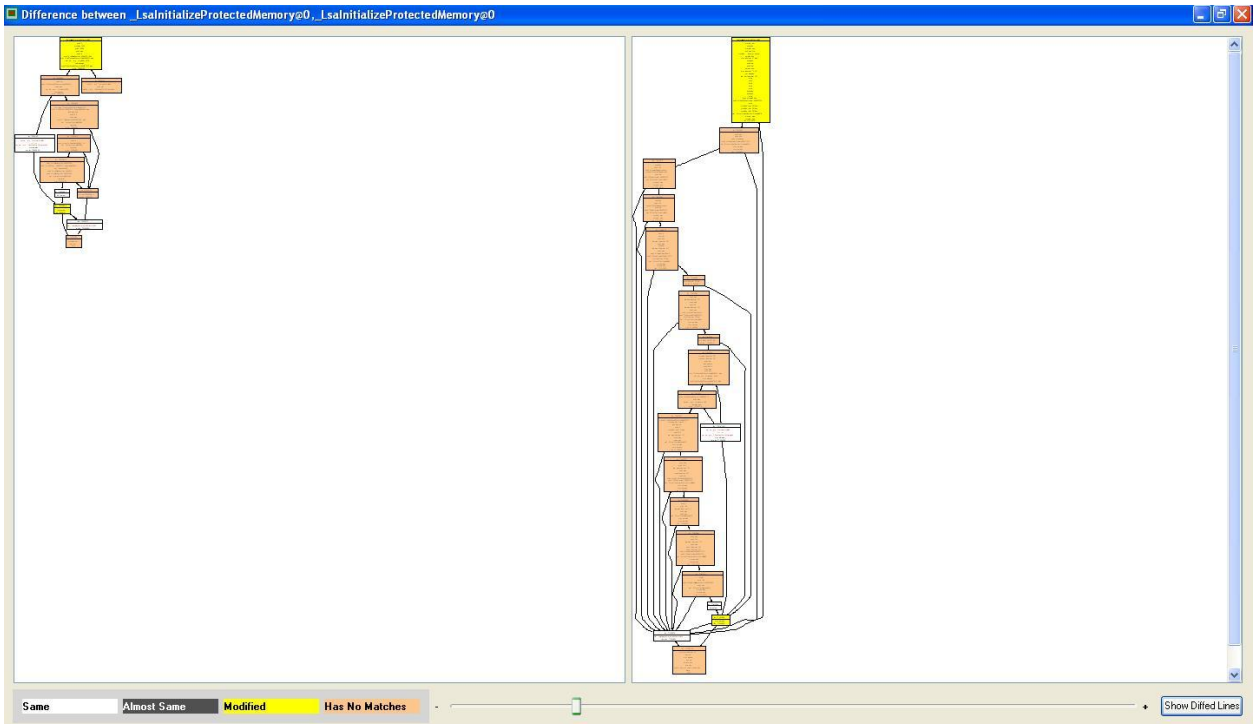


Figure 6 - LsaInitializeProtectedMemory() Comparison

Figure 6 shows the huge difference in execution flow for `LsaInitializeProtectedMemory()` between Vista and Vista SP1. Beyond the changes in raw execution flow, there are major changes in the functions and call structure used to generate the keys for encryption/decryption. The primary purpose of `LsaInitializeProtectedMemory()` is to create the corresponding encryption keys for use by `LsaEncryptMemory()`.

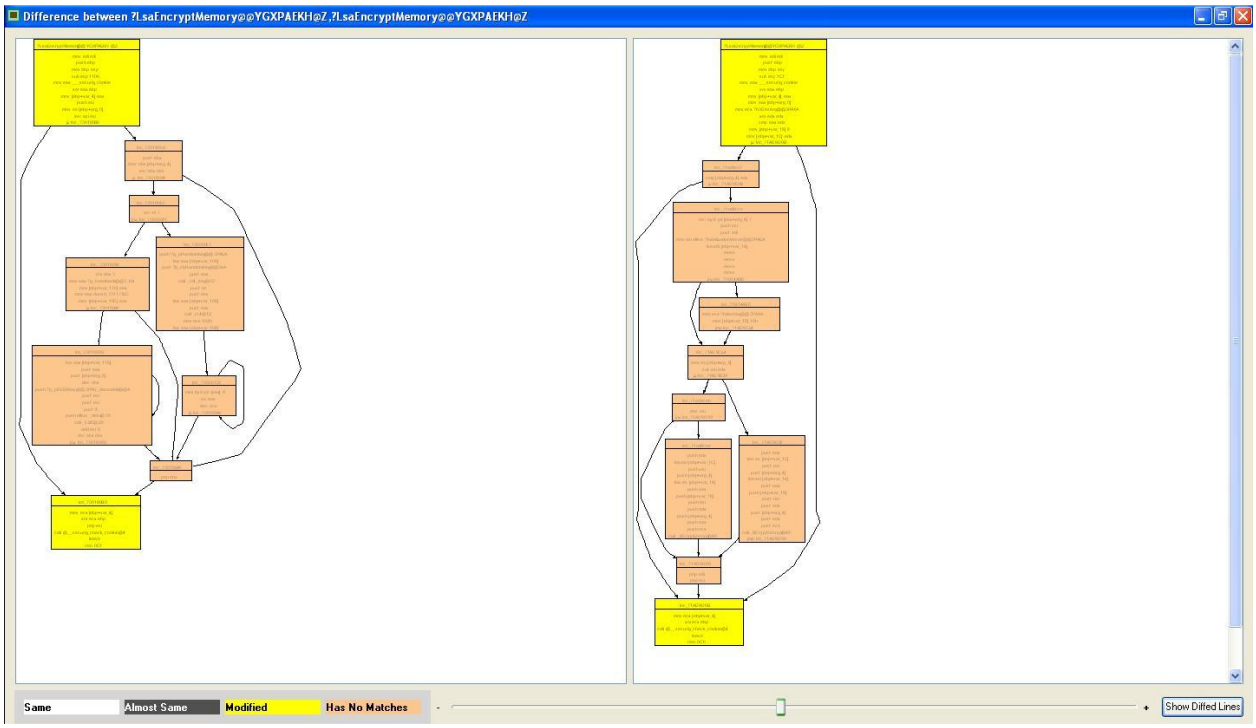


Figure 7 - LsaEncryptMemory() Comparison

A quick look at Figure 7 shows the differences in `LsaEncryptMemory()` do not appear as severe as `LsaInitializeProtectedMemory()`, but upon further inspection those differences becomes obvious. If you are not familiar with Darun Grim, the orange-brown colored boxes indicate pieces of code that have no match in the comparative function. The yellow boxes label code segments that have been modified. So you can see there is almost nothing in SP1 that exists in the RTM implementation of this function.

The next section of this paper will break out the changes in `LsaEncryptMemory()` and `LsaInitializeProtectedMemory()` in greater detail via disassembly and analysis of these functions to determine the specific code-level modifications made and to discover any potential paths to malicious techniques against the new modules.

Detailed Analysis

Here we will provide code level examples in C/C++ and assembly of the applicable functions contained with BCrypt.dll, and disassembly of the functions `LsaInitializeProtectedMemory()` and `LsaEncryptMemory()`. These examples will be broken down into three sub-sections:

1. Relevant Functions as Defined in BCrypt.h (CNG API)
2. Implementation examples of the relevant functions
3. Detailed analysis of `LsaInitializeProtectedMemory()` and `LsaEncryptMemory()`

Relevant Functions as Defined in BCrypt.h

BCrypt offers many functions as part of the CNG API. It is not necessary to examine all of these functions – we are only concerned with the ones used in the encryption/decryption of password hashes in memory. The specific functions we will examine are:

- BCryptOpenAlgorithmProvider
- BCryptSetProperty
- BCryptGetProperty
- BCryptEncrypt/Decrypt
- BCryptGenRandom
- BCryptGenerateSymmetricKey

Note: It is not necessary to go through all the code listings here. They are provided to give a full description of the functions, structures and some simple implementation examples of the functions we will examine from a reversing perspective. If you can follow reversing concepts and disassembly easily, it is not necessary to go through this section in great detail.

Code Listing 1 - BCryptOpenAlgorithmProvider

```
BCryptOpenAlgorithmProvider(
            __out       BCRYPT_ALG_HANDLE   *phAlgorithm,
            __in        LPCWSTR pszAlgId,
            __in_opt    LPCWSTR pszImplementation,
            __in        ULONG   dwFlags);
```

Code Listing 1: This function returns a handle to the opened primitive algorithm specified by the `pszAlgId` input.

Code Listing 2 - BCryptGetProperty

```
BCryptGetProperty(
  __in                                  BCRYPT_HANDLE   hObject,
  __in                                  LPCWSTR pszProperty,
  __out_bcount_part_opt(cbOutput, *pcbResult) PUCHAR   pbOutput,
  __in                                  ULONG   cbOutput,
  __out                                 ULONG   *pcbResult,
  __in                                  ULONG   dwFlags);
```

Code Listing 2: This function obtains the value for the specified property of an object – such as the size of a key object or block cipher length.

Code Listing 3 - BCryptSetProperty

```
BCryptSetProperty(
    __inout                 BCRYPT_HANDLE   hObject,
    __in                    LPCWSTR pszProperty,
    __in_bcount(cbInput)    PUCHAR   pbInput,
    __in                    ULONG   cbInput,
    __in                    ULONG   dwFlags);
```

Code Listing 3: This function sets the value of a specified object – such as selecting the mode for cipher block encryption.

Code Listing 4 - BCryptGenerateSymmetricKey

```
BCryptGenerateSymmetricKey(
    __inout                             BCRYPT_ALG_HANDLE   hAlgorithm,
    __out                               BCRYPT_KEY_HANDLE   *phKey,
    __out_bcount_full(cbKeyObject)      PUCHAR   pbKeyObject,
    __in                                ULONG   cbKeyObject,
    __in_bcount(cbSecret)               PUCHAR   pbSecret,
    __in                                ULONG   cbSecret,
    __in                                ULONG   dwFlags);
```

Code Listing 4: This function generates the symmetric key using the specified algorithm.

Code Listing 5 - BCryptEncrypt

```
BCryptEncrypt(
    __inout                                 BCRYPT_KEY_HANDLE hKey,
    __in_bcount(cbInput)                    PUCHAR   pbInput,
    __in                                    ULONG    cbInput,
    __in_opt                                VOID     *pPaddingInfo,
    __inout_bcount_opt(cbIV)                PUCHAR   pbIV,
    __in                                    ULONG    cbIV,
    __out_bcount_part_opt(cbOutput, *pcbResult) PUCHAR   pbOutput,
    __in                                    ULONG    cbOutput,
    __out                                   ULONG    *pcbResult,
    __in                                    ULONG    dwFlags);
```

Code Listing 5: This function performs the encryption .

Code Listing 6 - BCryptGenRandom

```
BCryptGenRandom(
    __inout                             BCRYPT_ALG_HANDLE   hAlgorithm,
    __inout_bcount_full(cbBuffer)       PUCHAR  pbBuffer,
    __in                                ULONG   cbBuffer,
    __in                                ULONG   dwFlags);
```

Code Listing 6: This function will fill a buffer with random bytes.

Implementation Examples

The following code listings are not complete – the purpose of these listings is to show some implementation snippets of the relevant cryptographic functions that we will be dissecting as we look at how Vista encrypts some in-memory values, such as password hashes. See the Microsoft CNG Development Kit or Windows SDK for more details.

Let's declare the necessary parameters we will need:

Code Listing 7 - Declarations

```
//declarations
BCRYPT_ALG_HANDLE       hAesProvider                = NULL;
BCRYPT_KEY_HANDLE       hAesKey                     = NULL;
NTSTATUS                status              = STATUS_UNSUCCESSFUL;
DWORD                   cbCipherText                = 0,
                        cbPlainText                 = 0,
                        cbData                      = 0,
                        cbKeyObject                 = 0,
                        cbBlockLen                  = 0,
                        cbBlob                      = 0;
PBYTE                   pbCipherText                = NULL,
                        pbPlainText                 = NULL,
                        pbKeyObject                 = NULL,
                        pbIV                        = NULL,
                        pbBlob                      = NULL
                        pbRandom                    = NULL;
```

Then open the provider that we want – in this example, it will be AES. This function returns a handle for the opened algorithm as `hAesProvider`.

Code Listing 8 - Open Algorithm

```
//open the algorithm handle
if(!NT_SUCCESS(status = BCryptOpenAlgorithmProvider(
                                &hAesProvider,
                                BCRYPT_AES_ALGORITHM,
                                NULL,
                                0)))
```

Once the primitive algorithm is opened you must then determine some sizes before you can do any work:

- Calculate the size of the blocks for the algorithm to be used (CBC, CFB, etc.) and the key object itself
- This is done via the `BCryptGetProperty()` function:

Code Listing 9 - Get Size of Key and Block Length

```
//calculate the size of the buffer for the KeyObject
if(!NT_SUCCESS(status = BCryptGetProperty(
                                    hAesProvider,
                                    BCRYPT_OBJECT_LENGTH,
                                    (PBYTE)&cbKeyObject,
                                    sizeof(DWORD),
                                    &cbData,
                                    0)))


//calculate the size of the block for IV
if(!NT_SUCCESS(status = BCryptGetProperty(
                                    hAesProvider,
                                    BCRYPT_BLOCK_LENGTH,
                                    (PBYTE)&cbBlockLen,
                                    sizeof(DWORD),
                                    &cbData,
                                    0)))
```

With the proper sizes of the key object and the appropriate block lengths, some size checking and buffer allocation would be in order (not shown here).

Then you would set the proper value for the key type you need; we are using AES CBC in this example. After setting this value, you would then generate the key.

Code Listing 10 - Set Value for Key type and Generate Key

```
//after some size checking and buffer allocation,
//set the value for the key to be created
if(!NT_SUCCESS(status = BCryptSetProperty(
                                hAesProvider,
                                BCRYPT_CHAINING_MODE,
                                (PBYTE)BCRYPT_CHAIN_MODE_CBC,
                                sizeof(BCRYPT_CHAIN_MODE_CBC),
                                0)))


// generate the key
if(!NT_SUCCESS(status = BCryptGenerateSymmetricKey(
                                 hAesProvider,
                                 &hAesKey,
                                 pbKeyObject, //key buffer
                                 cbKeyObject, //size of key buffer
                                 (PBYTE)myAESKey, //buffer with_
                                 sizeof(myAESKey), //secret key
                                 0)))
```

The `BCryptGenRandom()` function has many useful applications and is often used in cryptographic seeds, IVs and other purposes as needed. A quick example is given below:

Code Listing 11 - Random Generator

```
//Random generator - fills a buffer with random bytes
if(!NT_SUCCESS(status = BCryptGenRandom(
                                  hAesProvider,
                                  (PBYTE)&pbRandom, //pointer to_
                                  sizeof(pbRandom), //buffer
                                  0)))
```

Code Listing 12 - Encrypt and Decrypt functions

```
// save a copy of the key and IV for later use
// *** This is important because the BCryptEncrypt/Decrypt
// functions will alter them so that they cannot be reused ***
// We will not list this code here for brevity's sake


// use the key to encrypt the plaintext
if(!NT_SUCCESS(status = BCryptEncrypt(
                                    hAesKey, //key
                                    pbPlainText, //plain buffer
                                    cbPlainText, //size of buffer
                                    NULL, //NULL for symmetric
                                    pbIV, // buffer of IV
                                    cbBlockLen, //size of IV
                                    pbCipherText, //output buffer
                                    cbCipherText, //size of output
                                    &cbData, //pointer # bytes out
                                    BCRYPT_BLOCK_PADDING)))


// the decrypt function is listed as well
if(!NT_SUCCESS(status = BCryptDecrypt(
                                     hAesKey,
                                     pbCipherText,
                                     cbCipherText,
                                     NULL,
                                     pbIV,
                                     cbBlockLen,
                                     pbPlainText,
                                     cbPlainText,
                                     &cbPlainText,
                                     BCRYPT_BLOCK_PADDING)))
```

* The BCRYPT_BLOCK_PADDING is only used with symmetric algorithms and offloads manual calculations of block size multiples.

Comparative Analysis of `LsaInitializeProtectedMemory()` and `LsaEncryptMemory()` in Vista RTM and SP1

Now that we have a solid understanding of the relevant CNG structures and functions, let's dive into the analysis of how they are implemented within the Vista operating system directly.

The two functions within LSASRV.dll – `LsaInitializeProtectedMemory()` and `LsaEncryptMemory()` – are the primary functions involved in protecting in-memory hashes of Windows logon sessions. `LsaInitializeProtectedMemory()` generates the keys that will be used to perform the encryption and `LsaEncryptMemory()` actually performs the encryption and decryption of data.

We will be examining the significant differences in these functions between Vista SP1 and Vista RTM (and XP SP3 – I did not check previous versions, but suspect they will all be the same - or at least similar - since the "pash-the-hash" concept has applied to almost all previous NT-based OSs). At a high level, the protection mechanisms that encrypt in-memory password hashes have completely changed. Prior to Vista SP1, the primary mechanism used to perform the in-memory encryption was DES. Actually, it is a Microsoft specific implementation of DES called DESX (DES Extended).

In Vista SP1 (and Windows Server 2008), Microsoft has applied the new CNG functions to protect the in-memory hashes. The algorithms used to perform these functions are 3DES and AES.

In the Code Listings that will follow, please note the following :

```
Vista RTM code will be listed in blue text boxes
```

```
Vista SP1 code will be listed in red text boxes
```

Comments have been added throughout the code listings to make the analysis easier to follow.

Code Listing 13 - Disassembly and Analysis of RTM LsaInitializeProtectedMemory

VISTA RTM LsaInitializeProtectedMemory()

```
.text:  _LsaInitializeProtectedMemory@0 proc near
.text:
.text:  push    4
```

; set flProtect in VirtualAlloc(); 4 = Read/Write

```
.text:  mov     eax, 190h
.text:  push    1000h
```

; flAllocationType; 1000 = MEM_COMMIT – allocate memory and zero it

```
.text:  push    eax
.text:  push    0
.text:  mov     ?g_cbRandomKey@@3KA, 100h
```

; 256 bytes for cbRandomKey

```
.text:  mov     ?CredLockedMemorySize@@3KA, eax
.text:  call    ds:__imp__VirtualAlloc@16
```

; Call VirtualAlloc()

```
.text:  test    eax, eax
.text:  mov     ?CredLockedMemory@@3PAXA, eax
.text:  jz      loc_73056135
.text:  push    esi
.text:  push    ?CredLockedMemorySize@@3KA
.text:  push    eax
.text:  call    ds:__imp__VirtualLock@8
.text:  test    eax, eax
.text:  jz      loc_73056143
.text:  mov     eax, ?CredLockedMemory@@3PAXA
.text:  mov     ?g_pDESXKey@@3PAU_desxtable@@A, eax


.text:  add     eax, 90h
.text:  push    18h
.text:  push    eax
.text:  mov     ?g_pRandomKey@@3PAEA, eax
.text:  call    _SystemFunction036@8 ; SystemFunction036 is equivalent to
                                     ;ADVAPI32.dll!RtlGenRandom

.text:  test    al, al
.text:  jz      short loc_730348BF
.text:  push    8
.text:  push    offset ?g_Feedback@@3_KA
.text:  call    _SystemFunction036@8
.text:  test    al, al
.text:  jz      short loc_730348BF
.text:  push    ?g_pRandomKey@@3PAEA
.text:  push    ?g_pDESXKey@@3PAU_desxtable@@A
.text:  call    _desxkey@8
.text:  push    ?g_cbRandomKey@@3KA
.text:  push    ?g_pRandomKey@@3PAEA
.text:  call    _SystemFunction036@8
.text:  test    al, al
.text:  jz      short loc_730348BF
.text:  xor     esi, esi
```

; These lines are the Microsoft DES extended (DESX) algorithm that takes the random key, ;adds ;"whitening" padding from the desxtable and a Feedback value specified by g_Feedback; ;Then calls SystemFuntion036 to create the keys used for encryption - Note the use of ;g_pDESXKey, g_Feedback

```
.text: loc_730348BF:
.text:
.text:  mov     esi, 0C0000001h
.text:  jmp     loc_73056157
.text:  _LsaInitializeProtectedMemory@0 endp
```

Code Listing 14 - Disassembly and Analysis of SP1 LsaInitializeProtectedMemory

VISTA SP1 LsaInitializeProtectedMemory()

```
.text:  _LsaInitializeProtectedMemory@0 proc near

<…snip> ; variables and function startup stuff

.text:  stosd   ; "Store string" instruction – puts contents of EAX into address at EDI
                ; this is done four times (only showing one here)

.text:  push    ebx
.text:  push    ebx
.text:  stosw   ; "Store string" instruction for AX

.text:  push    offset a3des    ; "3DES"
.text:  push    offset ?h3DesProvider@@3PAXA
.text:  stosb   ; "Store string" instruction for AL

.text:  mov     [ebp+var_24], ebx
.text:  mov     [ebp+var_28], ebx
.text:  mov     [ebp+var_20], ebx
.text:  call    _BCryptOpenAlgorithmProvider@16

        ; First instance of call to new CNG Dll – Bcrypt.dll!BcryptOpenAlgorithmProvider
        ; Here the 3DES algorithm provider has been opened

.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      loc_71AF59FB
.text:  push    ebx
.text:  push    ebx
.text:  push    offset aAes     ; "AES"
.text:  push    offset ?hAesProvider@@3PAXA
.text:  call    _BCryptOpenAlgorithmProvider@16

        ; Open the AES algorithm provider

.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      loc_71AF59FB
.text:  push    ebx
.text:  push    20h
.text:  push    offset aChainingmodecb
.text:  mov     edi, offset aChainingmode
.text:  push    edi
.text:  push    ?h3DesProvider@@3PAXA
.text:  call    _BCryptSetProperty@20

        ; Set the 3DES provider to use the BCRYPT_CHAIN_MODE_CBC mode for symmetric keys

.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      loc_71AF59FB
.text:  push    ebx
.text:  push    20h
.text:  push    offset aChainingmodecf
.text:  push    edi
.text:  push    ?hAesProvider@@3PAXA
.text:  call    _BCryptSetProperty@20

        ; Set the AES provider to use the BCRYPT_CHAIN_MODE_CFB mode for symmetric keys

.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      loc_71AF59FB
.text:  push    4
.text:  pop     edi
.text:  push    ebx
.text:  lea     eax, [ebp+var_20]
.text:  push    eax
```

--= CONTINUED =--

```
.text:  push    edi
.text:  lea     eax, [ebp+var_24]
.text:  push    eax
.text:  push    offset aObjectlength
.text:  push    ?h3DesProvider@@3PAXA
.text:  mov     [ebp+var_20], edi
.text:  call    _BCryptGetProperty@24
```

; BCryptGetProperty – get the size of the key object for 3DES

```
.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      loc_71AF59FB
.text:  cmp     [ebp+var_20], edi
.text:  jnz     loc_71AF59E6
.text:  push    ebx
.text:  lea     eax, [ebp+var_20]
.text:  push    eax
.text:  push    edi
.text:  lea     eax, [ebp+var_28]
.text:  push    eax
.text:  push    offset aObjectlength
.text:  push    ?hAesProvider@@3PAXA
.text:  mov     [ebp+var_20], edi
.text:  call    _BCryptGetProperty@24
```

; BCryptGetProperty – get the size of the key object for AES

```
.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      loc_71AF59FB
.text:  cmp     [ebp+var_20], edi
.text:  jnz     loc_71AF59E6
.text:  mov     eax, [ebp+var_28]
.text:  mov     ecx, [ebp+var_24]

<…snip – VirtualAlloc init…>

.text:  test    eax, eax
.text:  jz      loc_71B14AC8
.text:  mov     edi, ?CredLockedMemory@@3PAXA
.text:  mov     eax, [ebp+var_24]
.text:  add     eax, edi
.text:  push    2
.text:  mov     [ebp+var_2C], eax
.text:  push    18h
.text:  lea     eax, [ebp+var_1C]
.text:  push    eax
.text:  push    ebx
.text:  call    _BCryptGenRandom@16
```

; BCryptGenRandom – fill a buffer with random bytes

```
.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      loc_71AF59FB
.text:  push    ebx
.text:  push    18h
.text:  lea     eax, [ebp+var_1C]
.text:  push    eax
.text:  push    [ebp+var_24]
.text:  push    edi
.text:  push    offset ?h3DesKey@@3PAXA
.text:  push    ?h3DesProvider@@3PAXA
.text:  call    _BCryptGenerateSymmetricKey@28
```

; Generate the Symmetric Key with 3DES

```
.text:  mov     esi, eax
.text:  cmp     esi, ebx
```

--= CONTINUED =--

```
.text:  jl      short loc_71AF59FB
.text:  push    2
.text:  push    10h
.text:  lea     eax, [ebp+var_1C]
.text:  push    eax
.text:  push    ebx
.text:  call    _BCryptGenRandom@16
.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      short loc_71AF59FB
.text:  push    ebx
.text:  push    10h
.text:  lea     eax, [ebp+var_1C]
.text:  push    eax
.text:  push    [ebp+var_28]
.text:  push    [ebp+var_2C]
.text:  push    offset ?hAesKey@@3PAXA
.text:  push    ?hAesProvider@@3PAXA
.text:  call    _BCryptGenerateSymmetricKey@28
```

; Generate the Symmetric Key with AES

```
.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      short loc_71AF59FB
.text:  push    2
.text:  push    10h
.text:  push    offset ?InitializationVector@@3PAEA
```

; reference the IV

```
.text:  push    ebx
.text:  call    _BCryptGenRandom@16
.text:  mov     esi, eax
.text:  cmp     esi, ebx
.text:  jl      short loc_71AF59FB
.text:  xor     esi, esi
.text:

<…snip…>

.text:
.text:  call    _LsaCleanupProtectedMemory@0
.text:  jmp     short loc_71AF59EA
.text:  _LsaInitializeProtectedMemory@0 endp
```

--= END =--

Code Listing 15 - Disassembly and Analysis of RTM LsaEncryptMemory

Vista RTM LsaEncryptMemory

```
.text: ?LsaEncryptMemory@@YGXPAEKH@Z proc near
.text:
.text: var_110         = dword ptr -110h
.text: var_10C         = dword ptr -10Ch
.text: var_108         = byte ptr -108h
.text: var_4           = dword ptr -4
.text: arg_0           = dword ptr  8
.text: arg_4           = dword ptr  0Ch
.text: arg_8           = dword ptr  10h

.text:  mov     edi, edi
.text:  push    ebp
.text:  mov     ebp, esp
.text:  sub     esp, 110h
.text:  mov     eax, ___security_cookie
.text:  xor     eax, ebp
.text:  mov     [ebp+var_4], eax
.text:  push    esi
.text:  mov     esi, [ebp+arg_0]
.text:  test    esi, esi
.text:  jz      short loc_730180B0
.text:  push    ebx
.text:  mov     ebx, [ebp+arg_4]
.text:  test    ebx, ebx
.text:  jz      short loc_730180AF
.text:  test    bl, 7
.text:  jnz     loc_730560F7
.text:  shr     ebx, 3
.text:  mov     eax, ?g_Feedback@@3_KA
        ; unsigned __int64 from LsaInitializeProtectedMemory

.text:  mov     [ebp+var_110], eax
.text:  mov     eax, dword_731171EC
.text:  mov     [ebp+var_10C], eax
.text:  jz      short loc_730180AF

.text:  lea     eax, [ebp+var_110]
.text:  push    eax
.text:  push    [ebp+arg_8]
.text:  dec     ebx
.text:  push    ?g_pDESXKey@@3PAU_desxtable@@A ; DES table; key plus whitening
.text:  push    esi
.text:  push    esi
.text:  push    8
.text:  push    offset _desx@16 ; DESX function
.text:  call    _CBC@28  ; Cipher block chain mode – performs the encryption
.text:  add     esi, 8
.text:  test    ebx, ebx
.text:  jnz     short loc_73018089
.text:
.text: loc_730180AF:
.text:
.text:  pop     ebx
.text:
.text: loc_730180B0:
.text:
.text:  mov     ecx, [ebp+var_4]
.text:  xor     ecx, ebp
.text:  pop     esi
.text:  call    @__security_check_cookie@4
.text:  leave
.text:  retn    0Ch
.text: ?LsaEncryptMemory@@YGXPAEKH@Z endp
```

Code Listing 16 - Disassembly and Analysis of SP1 LsaEncryptMemory

Vista SP1 LsaEncryptMemory

```
.text: ?LsaEncryptMemory@@YGXPAEKH@Z proc near
.text:
.text: var_1C          = dword ptr -1Ch
.text: var_18          = dword ptr -18h
.text: var_14          = byte ptr -14h       ; ptr for IV
.text: var_4           = dword ptr -4
.text: arg_0           = dword ptr  8        ; uchar *
.text: arg_4           = dword ptr  0Ch      ; ulong
.text: arg_8           = dword ptr  10h      ; int – 0 or 1
.text:
.text:
.text:  mov     edi, edi
.text:  push    ebp
.text:  mov     ebp, esp
.text:  sub     esp, 1Ch
.text:  mov     eax, ___security_cookie
.text:  xor     eax, ebp
.text:  mov     [ebp+var_4], eax
.text:  mov     eax, [ebp+arg_0]
.text:  mov     ecx, ?h3DesKey@@3PAXA
.text:  xor     edx, edx
.text:  cmp     eax, edx
.text:  mov     [ebp+var_18], 8
.text:  mov     [ebp+var_1C], edx
.text:  jz      short loc_71AD9D0B
.text:  cmp     [ebp+arg_4], edx
.text:  jz      short loc_71AD9D0B
.text:  test    byte ptr [ebp+arg_4], 7

        ; mask ulong arg_4 with 00000111

.text:  push    esi
.text:  push    edi
.text:  mov     esi, offset ?InitializationVector@@3PAEA
.text:  lea     edi, [ebp+var_14]
.text:  movsd           ; movsd = mov dword from ESI to EDI – here the IV is being moved into
                        ;the pointer specified by var_14. Four movsd means 128 bits
.text:  movsd
.text:  movsd
.text:  movsd
.text:  jnz     loc_71B14AB6  ; If not zero after 128 bits, Go to AES stuff
.text:
.text: loc_71AD9CE4:
.text:
.text:  mov     esi, [ebp+arg_8]
.text:  sub     esi, edx              ; Encrypt or Decrypt?
.text:  jz      short loc_71AD9D2F    ; 0 = Decrypt
.text:  dec     esi
.text:  jnz     short loc_71AD9D09
.text:  push    edx
.text:  lea     esi, [ebp+var_1C]
.text:  push    esi
.text:  push    [ebp+arg_4]
.text:  lea     esi, [ebp+var_14]
.text:  push    eax
.text:  push    [ebp+var_18]
.text:  push    esi
.text:  push    edx
.text:  push    [ebp+arg_4]
.text:  push    eax
.text:  push    ecx
.text:  call    _BCryptEncrypt@40     ; Encrypt data
```

--= CONTINUED =--

```
.text:
.text: loc_71AD9D09:
.text:
.text:  pop     edi
.text:  pop     esi
.text:
.text: loc_71AD9D0B:
.text:
.text:  mov     ecx, [ebp+var_4]
.text:  xor     ecx, ebp
.text:  call    @__security_check_cookie@4
.text:  leave
.text:  retn    0Ch
.text: ?LsaEncryptMemory@@YGXPAEKH@Z endp
```

; Jump location for the AES key

```
.text: ; ---------------------------------------------------
.text: loc_71B14AB6:
.text:   mov     ecx, ?hAesKey@@3PAXA
.text:   mov     [ebp+var_18], 10h
.text:   jmp     loc_71AD9CE4
.text: ; ---------------------------------------------------
```

; Jump location for the Decrypt function

```
.text: ; ---------------------------------------------------
.text: loc_71AD9D2F:
.text:  push    edx
.text:  lea     esi, [ebp+var_1C]
.text:  push    esi
.text:  push    [ebp+arg_4]
.text:  lea     esi, [ebp+var_14]
.text:  push    eax
.text:  push    [ebp+var_18]
.text:  push    esi
.text:  push    edx
.text:  push    [ebp+arg_4]
.text:  push    eax
.text:  push    ecx
.text:  call    _BCryptDecrypt@40     ; Decrypt data
.text:  jmp     short loc_71AD9D09
.text: ; ---------------------------------------------------
```

--= END =--

Observations of Analysis

It is obvious that Microsoft has significantly changed the implementation of algorithms to protect password hashes stored in memory. The DESX algorithm is no longer used for this purpose as of Vista SP1 and Windows Server 2008. Other changes are listed below:

- Most of the encryption process is self-contained in pre-SP1 code (LSASRV.dll); in SP1, LSASRV.dll makes external calls to BCrypt.dll to provide these functions
- There are no `SystemFunctionXXX()` calls in SP1 encryption/decryption of in-memory password hashes
- The CNG API provides all cryptographic functions for password hashes stored in memory in SP1

The new CNG API is implemented by BCrypt.dll primarily, with storage support provided by NCrypt.dll and kernel support implemented in Ksecdd.sys. The algorithms that provide protection of in memory password hashes in Vista SP1 3are DES and AES.

Although the algorithms have changed, the existing tools of the trade that could extract in memory password hashes *attacked the keys* or used DLL injection – can we do the same in Vista SP1? Is the implementation of the algorithms similar enough so that the same basic techniques can be applied to obtain the in memory hashes? DLL injection – particularly with LSASS – can be problematic due to the concept of Integrity Levels in Vista for system services. ASLR may also make this difficult. Another technique was to find the encryption key in memory and use that to just read the hashes right out of memory.

The corresponding presentation to this paper will demonstrate the possibility of obtaining the in-memory hashes by utilizing similar techniques as existing toolsets.

Challenges to Extracting the Hashes

Some of the challenges already identified to obtaining the password hashes stored in-memory on Vista SP1 are listed here:

- Different base address of LSASRV.dll on each boot (ASLR)
- Integrity Mechanism/Levels make DLL injection more difficult
- Different algorithms used for encryption/decryption – i.e. – cannot swap code and offsets from other tools
- Current research indicates the key and IV used during encryption of the hashes in memory are not available after the encrypt/decrypt function, or they are modified in a way that prevents direct re-use. This research is still on-going, however.

This author does not believe these challenges to be permanent and is looking for solutions to address these challenges. By sharing this information with the InfoSec community, perhaps others can discover solutions to these items, because it can be assumed that the malicious community has done/will do so soon.

APPENDIX

References & Tools:

Pass the Hash Toolkit – Hernan Ochoa, Core Security
- http://oss.coresecurity.com/projects/pshtoolkit.htm
- http://conference.hackinthebox.org/hitbsecconf2008kl/materials/D1T1%20-%20Hernan%20Ochoa%20-%20Pass-The-Hash%20Toolkit%20for%20Windows.pdf

Darun Grim Binary Diffing Suite – eEye Digital Security
- http://research.eeye.com/html/tools/RT20060801-1.html

Microsoft CNG SDK for Windows Vista and Windows Server 2008
- http://www.microsoft.com/downloads/details.aspx?familyid=1ef399e9-b018-49db-a98b-0ced7cb8ff6f&displaylang=en

MSDN CNG Article:
- http://msdn.microsoft.com/en-us/magazine/cc163389.aspx

CNG and Vista SP1:
- Random number generator not FIPS compliant: http://support.microsoft.com/kb/954059
- http://msdn.microsoft.com/en-us/library/bb204775(VS.85).aspx
- http://msdn.microsoft.com/en-us/library/aa375534(VS.85).aspx

Microsoft Windows Platform SDK
- http://msdn.microsoft.com/en-us/windowsserver/bb980924.aspx

Microsoft DDK/Windows Driver Kit
- http://www.microsoft.com/whdc/devtools/debugging/default.mspx

Windbg/Windows Debugging Tools
- http://www.microsoft.com/whdc/devtools/debugging/default.mspx

Windows Communications Protocols:
- http://msdn.microsoft.com/en-us/library/cc216513(PROT.10).aspx

IDA Pro
- http://www.hex-rays.com/idapro/

OpenRCE
- http://www.openrce.org

PEBrowsePro
- http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html