Windows Logon Password

Get Windows Logon Password using Wdigest in Memory Dump

2013-12-23

FOR - MD

Forensics MaJ3stY Deok9

Error is the discipline through which we advance. By William Ellery Channing

Keyword : Get Windows Logon Password, Wdigest, Mimikatz, Volatility Plugin

[1] Contact : saiwnsgud@gmail.com, http://maj3sty.tistory.com

1. Introduction

The former way to acquire the Windows logon password of user is to get a NTML hash value through the Windows logon session and registry then crack it. [Figure 1] shows the well-known ways to get a NTML hash value of user's windows logon password. For more information, take a look at "Dump Windows password hashes efficiently" on http://bernardodamele.blogspot.kr/.

Table 1 Way of obtaining NTLM hash of user's Windows logon password as per files

Files	Ways of obtaining NTLM hash of password
SAM	Decrypt the value of SAM hive file
NTDS.DIT	Decrypt after extracting the database table of NTDS.DIT
NTDS.DIT /SAM	Use the password history of NTDS.DIT/SAM hive file
SECURITY	Decrypt LSA secret of SECURITY hive file
SECURITY	Use the cached domain logon information of SECURITY hive file
MSV1.0	Use the credential information of Windows logon session

All of the obtained information using these methods is NTLM hash and it needs to be cracked with password crack tools. If the password is too long and even hard to crack, it is difficult to acquire the user's Windows logon password. However, the tool called "Mimikatz" [2] has been announced in 2012 to solve the problem. It uses DLL injection on live status so that it can print out the user's Windows logon password as a plaintext even though the password is long.

In this article, we'll apply one of the methods used in "Mimikatz" called "extracting user's Windows logon password using Wdigest" to memory dump, so we can help out the investigators with memory forensics.

[2] Mimikatz : http://blog.gentilkiwi.com/mimikatz

2. Windows Authentication Package

Windows Authentication Package is one of the major components to implement the Windows security and it includes Lsass process context and DLLs executed in client's process. The role of authentication DLL is to examine whether the user's name is in agreement with the password. If the authentication information is consistent, it returns the user's specific information to Lsass. Lsass create the token based on this. In typically, there are MSV1_0, TsPkg, Wdigest, LiveSSP, Kerberos, and SSP for windows authentication package and each package is carried out by various usage like Remote RDP, and Web service. It has a feature that it always carries the specific data in memory for Challenge-Response method. In this article, we will cover only Wdigest in Windows authentication package.

2.1 WDigest.dll

Wdigest.dll was first introduced in Windows XP system, and developed to authenticate the user in HTTP digest authentication and SASL (Simple Authentication Security Layer). This is used in digest authentication using Challenge-Response method as NTLM protocol. Also it transfers certificate through MD5 hash or message digest and it offers more improved security than before. However, to get a key for authentication, user's plaintext password is necessary and it can be abused. [Figure 1] and [Table 2] explan the digest authentication architecture and the elements.

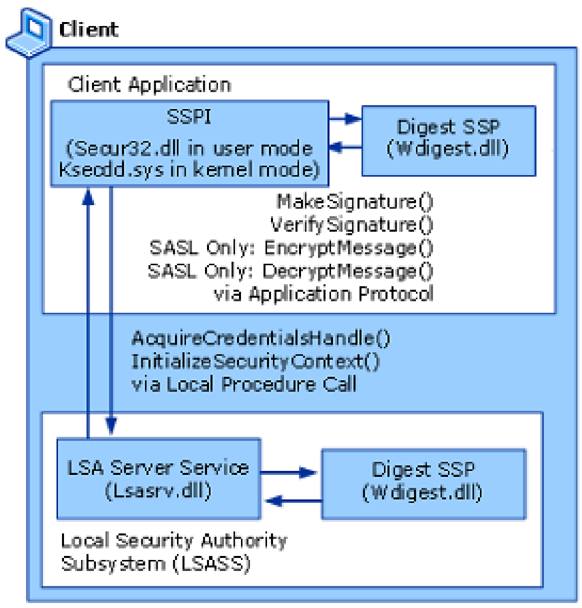


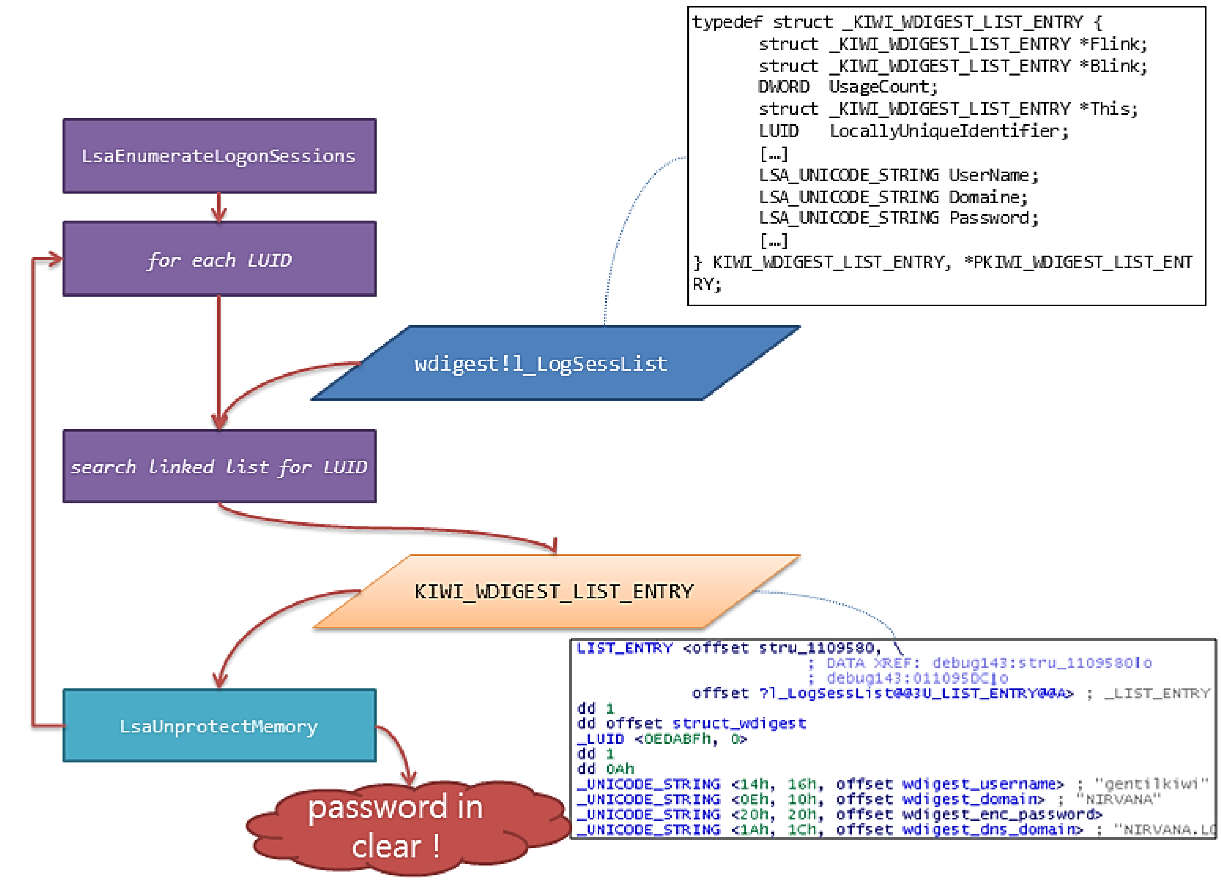
Figure 1 Digest Authentication Architecture

Table 2 Digest Authentication Elements

File	Explanation
Wdigest.dll	It works for SSP which is used for LDAP and Web authentication
Lsasrv.dll	Security service management of LSA (Security policy and behavior)
Secur32.dll	It works for application SSPI of user mode
Ksecdd.sys	It is used when kernel security device driver communicates with Lsass in user mode.

3. Extracting Windows logon password on live status

[Figure 2] shows the process of extraction of Windows logon password on live status using DLL injection in Wdigest.



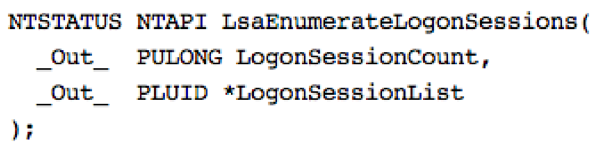
```
typedef struct _KIWI_WDIGEST_LIST_ENTRY {
        struct _KIWI_WDIGEST_LIST_ENTRY *Flink;
        struct _KIWI_WDIGEST_LIST_ENTRY *Blink;
        DWORD  UsageCount;
        struct _KIWI_WDIGEST_LIST_ENTRY *This;
        LUID   LocallyUniqueIdentifier;
        […]
        LSA_UNICODE_STRING UserName;
        LSA_UNICODE_STRING Domaine;
        LSA_UNICODE_STRING Password;
        […]
} KIWI_WDIGEST_LIST_ENTRY, *PKIWI_WDIGEST_LIST_ENT
RY;
```

```
LIST_ENTRY <offset stru_1109580, \
                        ; DATA XREF: debug143:stru_1109580|o
                        ; debug143:011095DC|o
            offset ?l_LogSessList@@3U_LIST_ENTRY@@A> ; _LIST_ENTRY
dd 1
dd offset struct_wdigest
_LUID <0EDABFh, 0>
dd 1
dd 0Ah
_UNICODE_STRING <14h, 16h, offset wdigest_username> ; "gentilkiwi"
_UNICODE_STRING <0Eh, 10h, offset wdigest_domain> ; "NIRVANA"
_UNICODE_STRING <20h, 20h, offset wdigest_enc_password>
_UNICODE_STRING <1Ah, 1Ch, offset wdigest_dns_domain> ; "NIRVANA.LO
```

Figure 2 The process of extraction of Windows logon password on live status using Wdigest

The working process for each element of conducted function/dll during extraction is as follows.

1. First, you have to collect the number of sessions and logon session identifiers (LUIDs) which exist in system through the LsaEnumerateLogonSessions function of Lsass. [Figure 3] is the LsaEnumerateLogonSessions function.



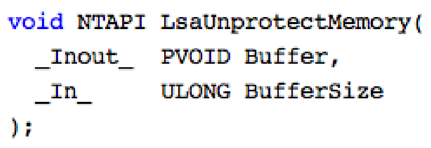
```
NTSTATUS NTAPI LsaEnumerateLogonSessions(
  _Out_  PULONG LogonSessionCount,
  _Out_  PLUID *LogonSessionList
);
```

Figure 3 LsaEnumerateLogonSessions Function

LogonSessionCount pointer variable has the number of logon sessions, and LogonSessionList pointer variable has the address value of the first element among the logon session identifiers. Based on this, you can trace the Logon session list existing in system.

2. I_LogSessList of Wdigest.dll is made of LIST_ENTRY structure, and it has use name, domain, encrypted password, domain DNS and so on written in Unicode character as well as Flink, Blink, and LUID.

3. Afterward, you can decrypt the encrypted password obtained from I_LogSessList through LsaUnprotectedMemory function of Lsasrv.dll. [Figure 4] shows the LsaUnprotectedMemory function[3].

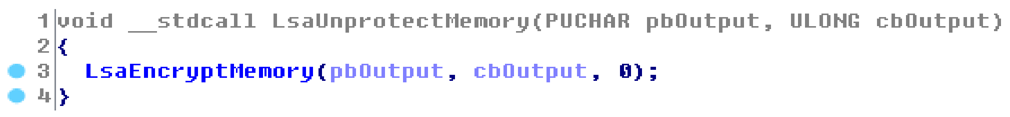


```
void NTAPI LsaUnprotectMemory(
  _Inout_  PVOID Buffer,
  _In_     ULONG BufferSize
);
```

Figure 4 LsaUnprotectedMemory Function

[3] http://msdn.microsoft.com/en-us/library/windows/desktop/ff714510(v=vs.85).aspx

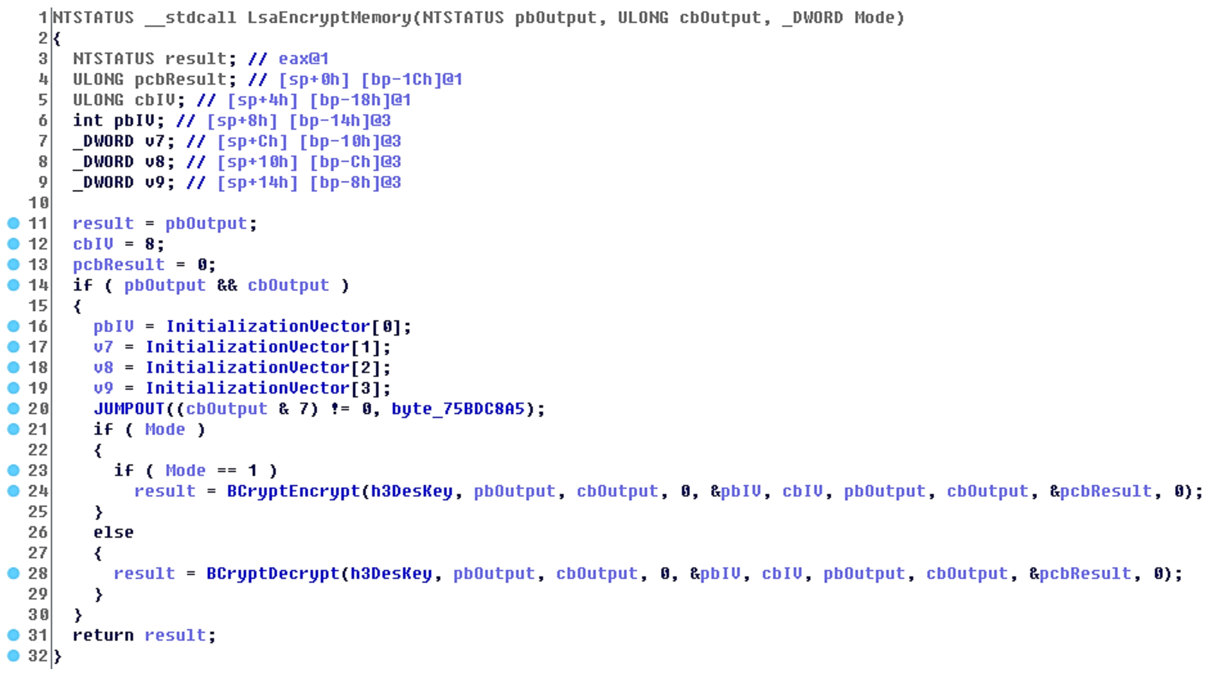
4. When the LsaUnProtectedMemory function is decompiled, it looks like [Figure 5].



```
void __stdcall LsaUnprotectMemory(PUCHAR pbOutput, ULONG cbOutput)
{
  LsaEncryptMemory(pbOutput, cbOutput, 0);
}
```

Figure 5 Decompile the LsaUnprotectMemory Function

It calls out LsaEncryptMemory function internally, and [Figure 6] suggests the result of decrypted LsaEncryptMemory function.



```
NTSTATUS __stdcall LsaEncryptMemory(NTSTATUS pbOutput, ULONG cbOutput, _DWORD Mode)
{
  NTSTATUS result; // eax@1
  ULONG pcbResult; // [sp+0h] [bp-1Ch]@1
  ULONG cbIV; // [sp+4h] [bp-18h]@1
  int pbIV; // [sp+8h] [bp-14h]@3
  _DWORD v7; // [sp+Ch] [bp-10h]@3
  _DWORD v8; // [sp+10h] [bp-Ch]@3
  _DWORD v9; // [sp+14h] [bp-8h]@3

  result = pbOutput;
  cbIV = 8;
  pcbResult = 0;
  if ( pbOutput && cbOutput )
  {
    pbIV = InitializationVector[0];
    v7 = InitializationVector[1];
    v8 = InitializationVector[2];
    v9 = InitializationVector[3];
    JUMPOUT((cbOutput & 7) != 0, byte_75BDC8A5);
    if ( Mode )
    {
      if ( Mode == 1 )
        result = BCryptEncrypt(h3DesKey, pbOutput, cbOutput, 0, &pbIV, cbIV, pbOutput, cbOutput, &pcbResult, 0);
    }
    else
    {
      result = BCryptDecrypt(h3DesKey, pbOutput, cbOutput, 0, &pbIV, cbIV, pbOutput, cbOutput, &pcbResult, 0);
    }
  }
  return result;
}
```

Figure 6 Decompile the LsaEncryptMemory function

With the result above, you can finally figure out that the LsaEncryptMemory function of Lsasrv.dll decrypt the encrypted password using pbIV and 3DesKey handle value.

We tried to explain the whole working process so far. Anyone who wants to check out the working process through code, you can visit https://github.com/thomhastings/mimikatz-en/.

4. Extracting Windows Logon Information in memory dump

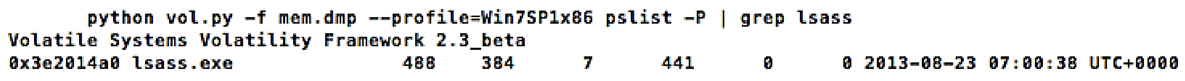
Before you extract the information, we will explain how to obtain windows logon password using WDigest in memory dump. Let's take a look at [Table 3].

Table 3 What you need to know

Category	Explanation
dll needed	WDigest.dll : It has the address of encrypted password value. Lsasrv.dll : It is needed to decrypt the encrypted password.
value to find	WDigest.dll : The address of encrypted password value of l_LogSessList Lsasrv.dll : The handle value of 3DesKey and pbIV value

When you only refer to the contents of [Table 3], extracting password may look simple. However it has to go under complicated order to find and trace the value in memory dump. You can only access the memory dump file we have by physical address because the pointer value of dll is based on virtual address.

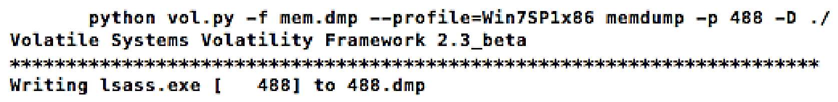
From now on, we will figure out how to extract the Windows Logon password in memory dump. First of all, you have to check out the parent process called PID of Lsass.exe to extract WDigest.dll and Lsasrv.dll. [Figure 7] shows the result of PID of Lsass.exe using pslist plugin of Volatility.



```
       python vol.py -f mem.dmp --profile=Win7SP1x86 pslist -P | grep lsass
Volatile Systems Volatility Framework 2.3_beta
0x3e2014a0 lsass.exe               488    384      7      441      0      0 2013-08-23 07:00:38 UTC+0000
```

Figure 7 Check the PID of Lsass.exe

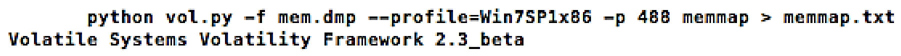
When Lsass.exe is being executed, it obtains the memory dump of Lsass.exe using memdump plugin to identify the value of allocated memory space. [Figure 8] shows the result of memory dump command of Lsass.exe, called PID488.



```
       python vol.py -f mem.dmp --profile=Win7SP1x86 memdump -p 488 -D ./
Volatile Systems Volatility Framework 2.3_beta
************************************************************************
Writing lsass.exe [   488] to 488.dmp
```

Figure 8 execution of Lsass.exe memory dump

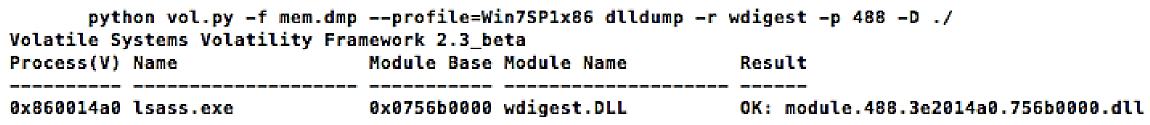
So far, we have tried to reduced the size of dump file we need to analyze to obtain the Windows Logon password by Lsass.exe memory dump, which has "whole memory dump -> every value to extract". As we mentioned, Lsass.exe memory dump also can be accessed by physical address. So you have to create the memory map file for mapping of virtual address and physical address. [Figure 9] shows how to extract memory map of relevant memory dump with memmap plugin.



```
       python vol.py -f mem.dmp --profile=Win7SP1x86 -p 488 memmap > memmap.txt
Volatile Systems Volatility Framework 2.3_beta
```

Figure 9 Collecting Memory Map

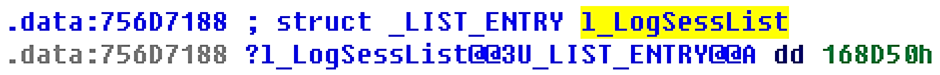
Next, you have to dump WDigest.dll, one of the dlls needed to extract Windows Logon password. [Figure 10] shows the command how to dump Wdigest.dll.



```
       python vol.py -f mem.dmp --profile=Win7SP1x86 dlldump -r wdigest -p 488 -D ./
Volatile Systems Volatility Framework 2.3_beta
Process(V) Name                 Module Base Module Name          Result
---------- -------------------- ----------- -------------------- ------
0x860014a0 lsass.exe            0x0756b0000 wdigest.DLL          OK: module.488.3e2014a0.756b0000.dll
```

Figure 10 WDigest.dll Dump

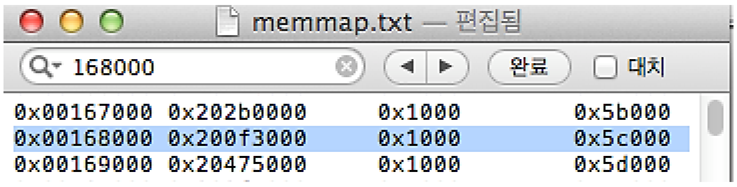
WDigest.dll has an address of encrypted password value, and you can check the very beginning address of the list out on I_LogSessList to trace it. It is made of LIST_ENTRY structure. [Figure 11] shows the stored value on I_LogSessList.



```
.data:756D7188 ; struct _LIST_ENTRY l_LogSessList
.data:756D7188 ?l_LogSessList@@3U_LIST_ENTRY@@A dd 168D50h
```

Figure 11 Identifying l_LogSessList

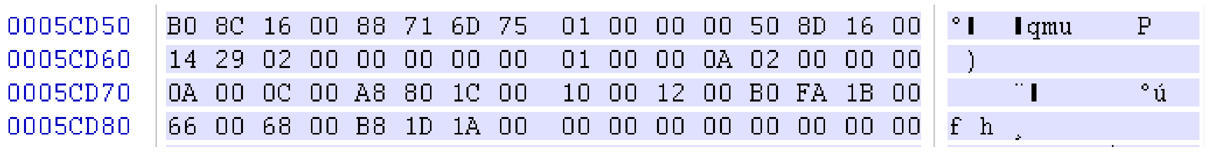
0x168D50 means the virtual address of the very first element of user's logon session list. You have to find the space including the relevant address, and then identify the physical address of Lsass.exe memory dump file. [Figure 12] shows the searching of virtual address space including 0x168D50 in memory map file.



```
0x00167000 0x202b0000     0x1000        0x5b000
0x00168000 0x200f3000     0x1000        0x5c000
0x00169000 0x20475000     0x1000        0x5d000
```

Figure 12 Searching the address including 0x168D50

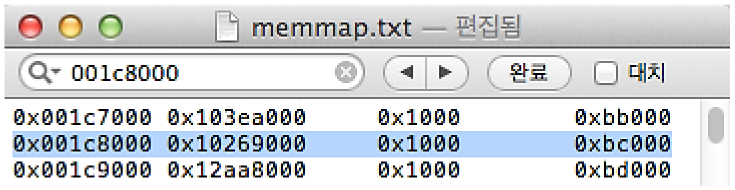
The space from size 0x00168000 to 0x1000 is mapped from 0x5c000 of Lsass.exe memory dump file. You can check the value by moving to 0x5C00 + (0x168D50 - 0x168D00) = 0x5CD50 from Lsass.exe physical memory dump. [Figure 13] shows the point of offset 0x5CD50 on WinHex.



```
0005CD50  B0 8C 16 00 88 71 6D 75  01 00 00 00 50 8D 16 00   °▌   ▌qmu     P
0005CD60  14 29 02 00 00 00 00 00  01 00 00 0A 02 00 00 00    )
0005CD70  0A 00 0C 00 A8 80 1C 00  10 00 12 00 B0 FA 1B 00       ¨▌      °ú
0005CD80  66 00 68 00 B8 1D 1A 00  00 00 00 00 00 00 00 00   f h ¸
```

Figure 13 The Point of 0x5CD50 in Lsass.exe physical memory dump

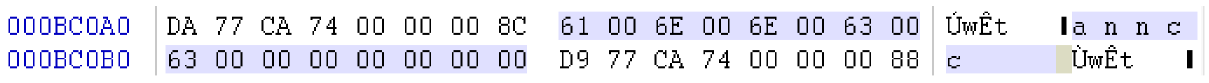
On 0x5CD50, you can find the user account on 0x20. If you cannot find anything, you have to check the next 4 byte. In this memory dump, the user account is on the point 0x001C80A9. This point is 4 byte away from offset 0x5CD74. You can check it out in [Figure 14].



```
0x001c7000 0x103ea000     0x1000        0xbb000
0x001c8000 0x10269000     0x1000        0xbc000
0x001c9000 0x12aa8000     0x1000        0xbd000
```

Figure 14 Searching the address including 0x001C80A8

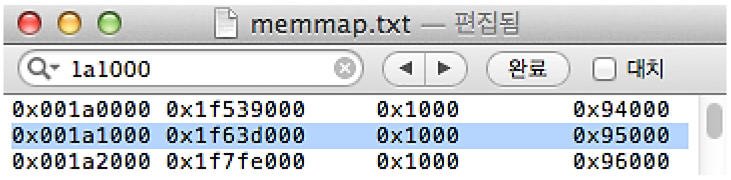
Now let's move to point 0xBC000 + (0x1C80A8 - 0x1C8000) = 0xBC0A8 in Lsass.exe memory dump.You can identify the user account by unicode. This is shown in [Figure 15].



```
000BC0A0  DA 77 CA 74 00 00 00 8C  61 00 6E 00 6E 00 63 00  ÚwÊt    ▌a n n c
000BC0B0  63 00 00 00 00 00 00 00  D9 77 CA 74 00 00 00 88  c       ÙwÊt   ▌
```

Figure 15 Identifying User Account

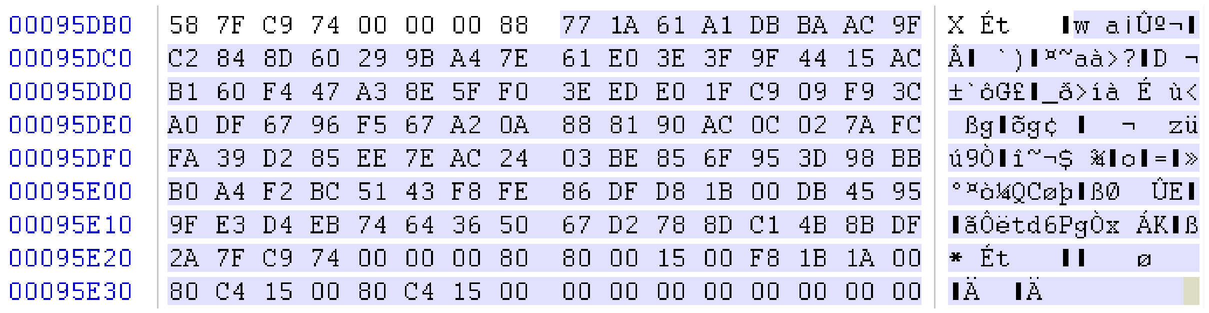
The encrypted Windows Logon password of user is on offset 0x5CD74 + 0x10, which has the user's account information address. You can check the value of physical address 0x001A1DB8 of point 0x5CD84 in [Figure 13] like shown in [Figure 16],



```
0x001a0000 0x1f539000      0x1000          0x94000
0x001a1000 0x1f63d000      0x1000          0x95000
0x001a2000 0x1f7fe000      0x1000          0x96000
```

Figure 16 Searching The Address including 0x001A1DB8

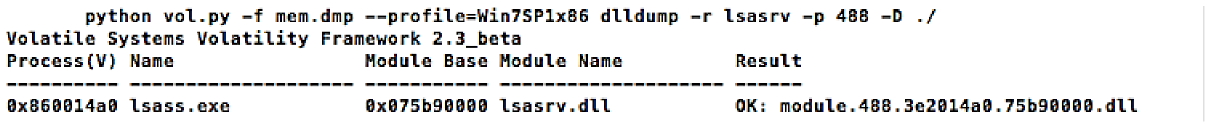
If you move to point 0x95DB8 in Lsass.exe memory dump, you can find the hex value shown in [Figure 17]. That part refers to the encrypted user's password achievable on WDigest. Because there is 0x00(NULL) in the middle, you can figure out that the actual valid value of encrypted user's Windows logon password is by 0x95E23.



```
00095DB0  58 7F C9 74 00 00 00 88  77 1A 61 A1 DB BA AC 9F  X Ét    ▌w a¡Ûº¬▌
00095DC0  C2 84 8D 60 29 9B A4 7E  61 E0 3E 3F 9F 44 15 AC  Â▌ `)▌¤~aà>?▌D ¬
00095DD0  B1 60 F4 47 A3 8E 5F F0  3E ED E0 1F C9 09 F9 3C  ±`ôG£▌_ð>íà É ù<
00095DE0  A0 DF 67 96 F5 67 A2 0A  88 81 90 AC 0C 02 7A FC   ßg▌õg¢ ▌   ¬  zü
00095DF0  FA 39 D2 85 EE 7E AC 24  03 BE 85 6F 95 3D 98 BB  ú9Ò▌î~¬$ ¾▌o▌=▌»
00095E00  B0 A4 F2 BC 51 43 F8 FE  86 DF D8 1B 00 DB 45 95  °¤ò¼QCøþ▌ßØ  ÛE▌
00095E10  9F E3 D4 EB 74 64 36 50  67 D2 78 8D C1 4B 8B DF  ▌ãÔëtd6PgÒx ÁK▌ß
00095E20  2A 7F C9 74 00 00 00 80  80 00 15 00 F8 1B 1A 00  * Ét   ▌▌   ø
00095E30  80 C4 15 00 80 C4 15 00  00 00 00 00 00 00 00 00  ▌Ä  ▌Ä
```

Figure 17 Encrypted User's Windows Logon Password

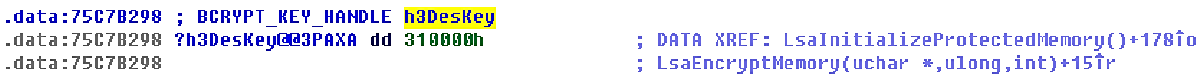
For next, you have to dump the Lsasrv.dll which is necessary to decrypt the user's encrypted Windows logon password. [Figure 18] shows the result of dumped dll. The way of plugin command is same as WDigest.dll.



```
       python vol.py -f mem.dmp --profile=Win7SP1x86 dlldump -r lsasrv -p 488 -D ./
Volatile Systems Volatility Framework 2.3_beta
Process(V) Name                 Module Base Module Name          Result
---------- -------------------- ----------- -------------------- ------
0x860014a0 lsass.exe            0x075b90000 lsasrv.dll           OK: module.488.3e2014a0.75b90000.dll
```

Figure 18 Dump of Lsasrv.dll

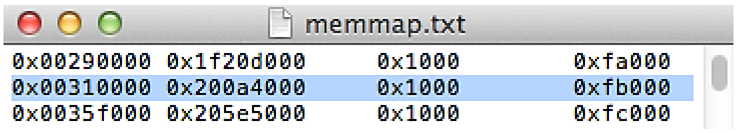
The handle value of 3DesKey is used during the process of decryption of LsaEncryptMemory function, and it is shown in [Figure 19]. However, what we actually need is the value of pbSecret instead of the handle value of 3DesKey. Because the final values to decrypt are composed of pbSecret, pbIV, and encrypted user's Windows logon password. The value of pbSecret exists in the address of 3DesKey and the location of 0x3C.



```
.data:75C7B298 ; BCRYPT_KEY_HANDLE h3DesKey
.data:75C7B298 ?h3DesKey@@3PAXA dd 310000h                ; DATA XREF: LsaInitializeProtectedMemory()+178îo
.data:75C7B298                                            ; LsaEncryptMemory(uchar *,ulong,int)+15îr
```

Figure 19 Address of 3DesKey

By [Figure 20], you can see that 0x31000 is mapped to the physical address of 0xFB000.

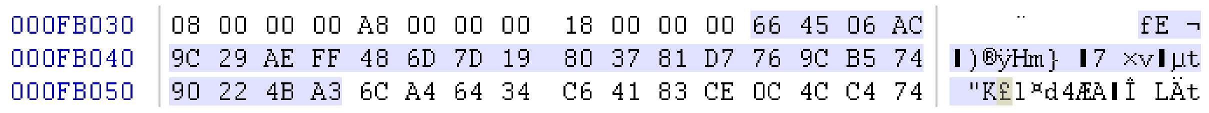


memmap.txt

```
0x00290000 0x1f20d000     0x1000      0xfa000
0x00310000 0x200a4000     0x1000      0xfb000
0x0035f000 0x205e5000     0x1000      0xfc000
```

Figure 20 Searching the address including 0x310000

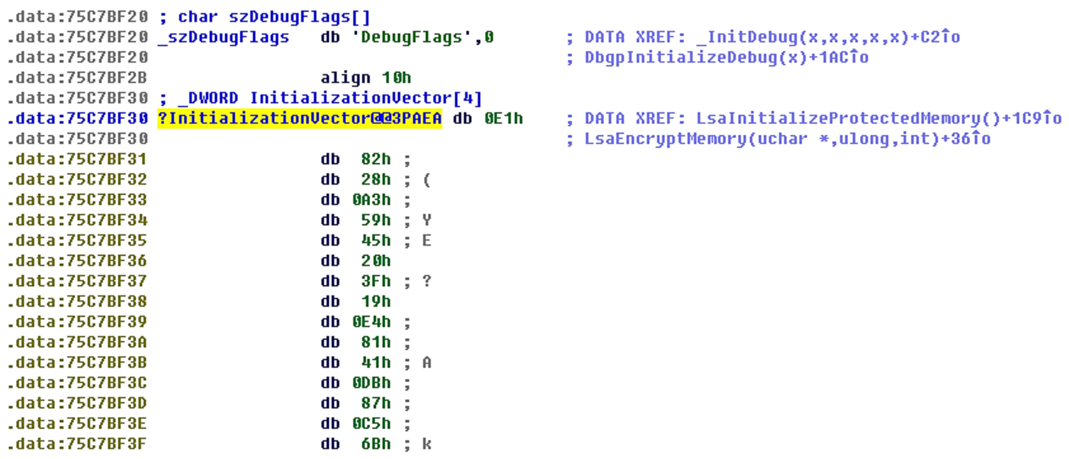
As we mentioned, pbSecret exists right after the location of 0x3C. Therefore you need to move to 0XFB000 + 0x3C = 0xFB03C of Lsass.exe memory dump. As [Figure 21] shows, there is room of 4 byte in front of the address, and you can find the hex value right after that.



```
000FB030 | 08 00 00 00 A8 00 00 00  18 00 00 00 66 45 06 AC |    ¨       fE ¬
000FB040 | 9C 29 AE FF 48 6D 7D 19  80 37 81 D7 76 9C B5 74 | ▌)®ÿHm} ▌7 ×v▌µt
000FB050 | 90 22 4B A3 6C A4 64 34  C6 41 83 CE 0C 4C C4 74 |  "K£l¤d4ÆA▌Î LÄt
```

Figure 21 The value of pbSecret

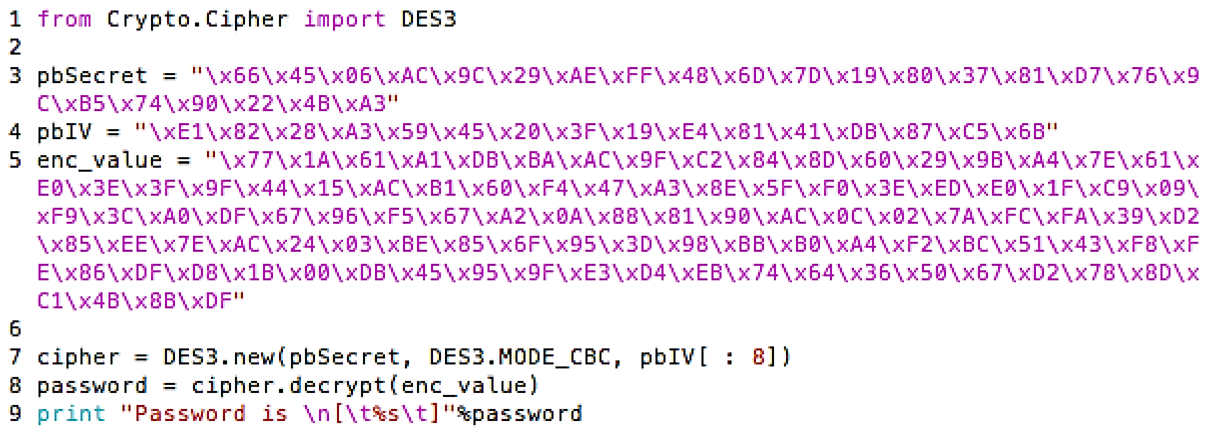
At last, you can figure out the pbIV value with just checking the first index value of array InitializationVector[4] of Lsasrv.dll. [Figure 22] shows the result of value of pbIV.



```
.data:75C7BF20 ; char szDebugFlags[]
.data:75C7BF20 _szDebugFlags   db 'DebugFlags',0       ; DATA XREF: _InitDebug(x,x,x,x,x)+C2↑o
.data:75C7BF20                                         ; DbgpInitializeDebug(x)+1AC↑o
.data:75C7BF2B                 align 10h
.data:75C7BF30 ; _DWORD InitializationVector[4]
.data:75C7BF30 ?InitializationVector@@3PAEA db 0E1h    ; DATA XREF: LsaInitializeProtectedMemory()+1C9↑o
.data:75C7BF30                                         ; LsaEncryptMemory(uchar *,ulong,int)+36↑o
.data:75C7BF31                 db  82h ;
.data:75C7BF32                 db  28h ; (
.data:75C7BF33                 db 0A3h ;
.data:75C7BF34                 db  59h ; Y
.data:75C7BF35                 db  45h ; E
.data:75C7BF36                 db  20h
.data:75C7BF37                 db  3Fh ; ?
.data:75C7BF38                 db  19h
.data:75C7BF39                 db 0E4h ;
.data:75C7BF3A                 db  81h ;
.data:75C7BF3B                 db  41h ; A
.data:75C7BF3C                 db 0DBh ;
.data:75C7BF3D                 db  87h ;
.data:75C7BF3E                 db 0C5h ;
.data:75C7BF3F                 db  6Bh ; k
```

Figure 22 The value of pbIV

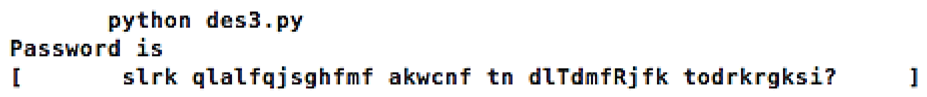
Last, you can check the actual user's logon password through the 3Des and obtained value using Python. [Figure 23] is shows the relevant Python code.



```
from Crypto.Cipher import DES3

pbSecret = "\x66\x45\x06\xAC\x9C\x29\xAE\xFF\x48\x6D\x7D\x19\x80\x37\x81\xD7\x76\x9
C\xB5\x74\x90\x22\x4B\xA3"
pbIV = "\xE1\x82\x28\xA3\x59\x45\x20\x3F\x19\xE4\x81\x41\xDB\x87\xC5\x6B"
enc_value = "\x77\x1A\x61\xA1\xDB\xBA\xAC\x9F\xC2\x84\x8D\x60\x29\x9B\xA4\x7E\x61\x
E0\x3E\x3F\x9F\x44\x15\xAC\xB1\x60\xF4\x47\xA3\x8E\x5F\xF0\x3E\xED\xE0\x1F\xC9\x09\
xF9\x3C\xA0\xDF\x67\x96\xF5\x67\xA2\x0A\x88\x81\x90\xAC\x0C\x02\x7A\xFC\xFA\x39\xD2
\x85\xEE\x7E\xAC\x24\x03\xBE\x85\x6F\x95\x3D\x98\xBB\xB0\xA4\xF2\xBC\x51\x43\xF8\xF
E\x86\xDF\xD8\x1B\x00\xDB\x45\x95\x9F\xE3\xD4\xEB\x74\x64\x36\x50\x67\xD2\x78\x8D\x
C1\x4B\x8B\xDF"

cipher = DES3.new(pbSecret, DES3.MODE_CBC, pbIV[ : 8])
password = cipher.decrypt(enc_value)
print "Password is \n[\t%s\t]"%password
```

Figure 23 Python code to decrypt the user's password

[Figure 24] shows the result of executed code. You can decrypt the password even it is very long.



```
        python des3.py
Password is
[       slrk qlalfqjsghfmf akwcnf tn dlTdmfRjfk todrkrgksi?      ]
```

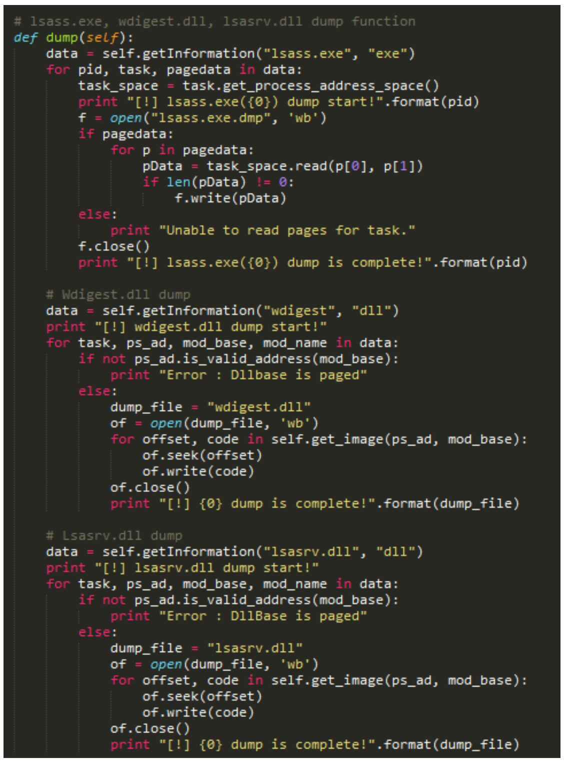
Figure 24 The result of executed code

5. Volatility Plugin - logon

We created the Volatility plugin based on the method of extracting the Windows logon password from memory, which we introduced in this article before. In this chapter, we will briefly cover how the plugin works. Let's take a look at the working order of plugin.

5.1. Memory dump of Lsass.exe, Wdigest.dll, and Lsasrv.dll

First, you need to extract some space to collect the most necessary information. In this plugin, we will dump the memory space of the process which owns the necessary space. [Figure 25] shows the function executing the dumping.



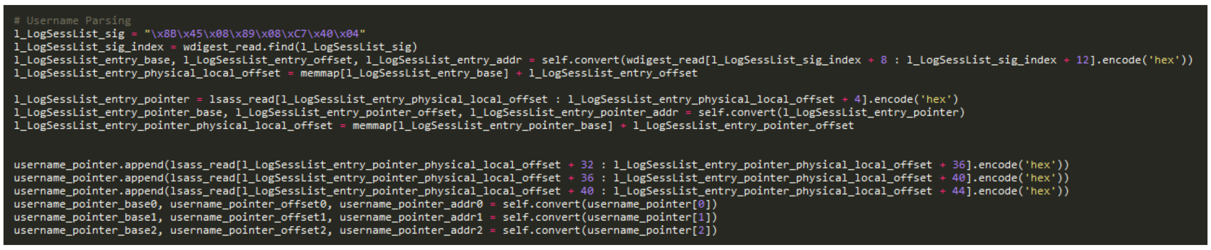
```python
# lsass.exe, wdigest.dll, lsasrv.dll dump function
def dump(self):
    data = self.getInformation("lsass.exe", "exe")
    for pid, task, pagedata in data:
        task_space = task.get_process_address_space()
        print "[!] lsass.exe({0}) dump start!".format(pid)
        f = open("lsass.exe.dmp", 'wb')
        if pagedata:
            for p in pagedata:
                pData = task_space.read(p[0], p[1])
                if len(pData) != 0:
                    f.write(pData)
        else:
            print "Unable to read pages for task."
        f.close()
        print "[!] lsass.exe({0}) dump is complete!".format(pid)

    # Wdigest.dll dump
    data = self.getInformation("wdigest", "dll")
    print "[!] wdigest.dll dump start!"
    for task, ps_ad, mod_base, mod_name in data:
        if not ps_ad.is_valid_address(mod_base):
            print "Error : Dllbase is paged"
        else:
            dump_file = "wdigest.dll"
            of = open(dump_file, 'wb')
            for offset, code in self.get_image(ps_ad, mod_base):
                of.seek(offset)
                of.write(code)
            of.close()
            print "[!] {0} dump is complete!".format(dump_file)

    # Lsasrv.dll dump
    data = self.getInformation("lsasrv.dll", "dll")
    print "[!] lsasrv.dll dump start!"
    for task, ps_ad, mod_base, mod_name in data:
        if not ps_ad.is_valid_address(mod_base):
            print "Error : DllBase is paged"
        else:
            dump_file = "lsasrv.dll"
            of = open(dump_file, 'wb')
            for offset, code in self.get_image(ps_ad, mod_base):
                of.seek(offset)
                of.write(code)
            of.close()
            print "[!] {0} dump is complete!".format(dump_file)
```

Figure 25 dump() function

Each necessary image can be dumped by basic dump modules provided by Volatility. You can use the name of the images you need from the list so to automatically dump with filtering.

5.2. Extracting the name of user's account

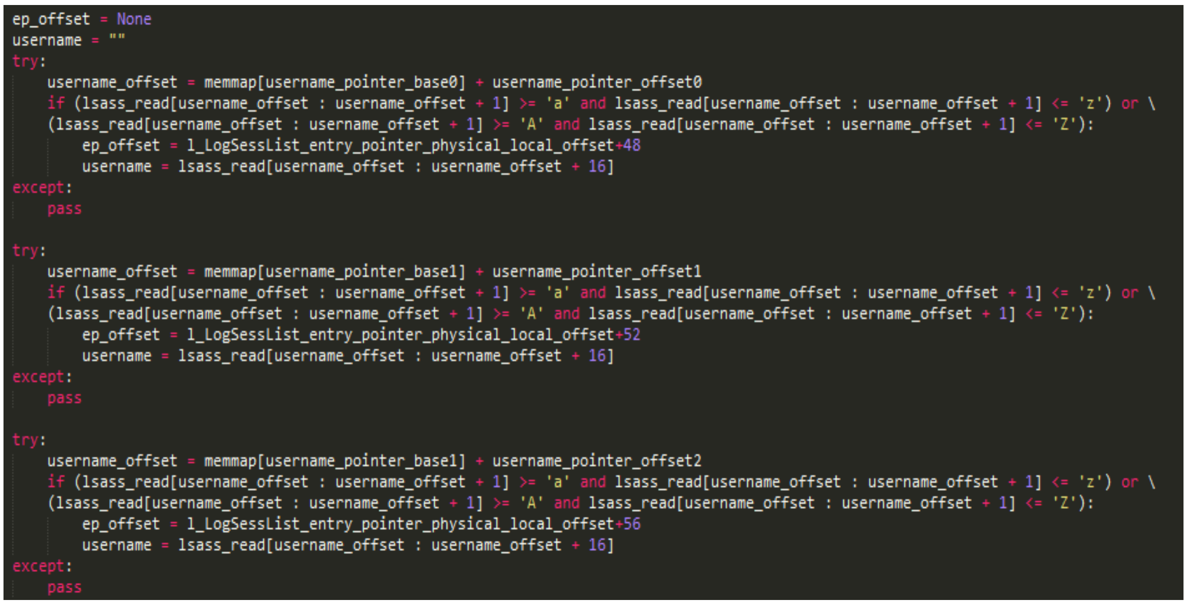
When extracting the name of user's account from memory, you need to check the value of l_LogSessList+0x20(the address which has the name of user's account as unicode). If the address is invalid, you have to check the address of 4 byte field to find the account. However, it is difficult to figure out which address has an account in which filed in plugin. Therefore you need to check the address value of specific space(4 byte * 3 times) first, and then move to that address to check if there is a valid character string and check if there is an account. When verifying a valid account, you have to follow the policy of Windows account name. [Figure 26] shows the routine to find the address which has account name.



```
# Username Parsing
l_LogSessList_sig = "\x8B\x45\x08\x89\x08\xC7\x40\x04"
l_LogSessList_sig_index = wdigest_read.find(l_LogSessList_sig)
l_LogSessList_entry_base, l_LogSessList_entry_offset, l_LogSessList_entry_addr = self.convert(wdigest_read[l_LogSessList_sig_index + 8 : l_LogSessList_sig_index + 12].encode('hex'))
l_LogSessList_entry_physical_local_offset = memmap[l_LogSessList_entry_base] + l_LogSessList_entry_offset

l_LogSessList_entry_pointer = lsass_read[l_LogSessList_entry_physical_local_offset : l_LogSessList_entry_physical_local_offset + 4].encode('hex')
l_LogSessList_entry_pointer_base, l_LogSessList_entry_pointer_offset, l_LogSessList_entry_pointer_addr = self.convert(l_LogSessList_entry_pointer)
l_LogSessList_entry_pointer_physical_local_offset = memmap[l_LogSessList_entry_pointer_base] + l_LogSessList_entry_pointer_offset


username_pointer.append(lsass_read[l_LogSessList_entry_pointer_physical_local_offset + 32 : l_LogSessList_entry_pointer_physical_local_offset + 36].encode('hex'))
username_pointer.append(lsass_read[l_LogSessList_entry_pointer_physical_local_offset + 36 : l_LogSessList_entry_pointer_physical_local_offset + 40].encode('hex'))
username_pointer.append(lsass_read[l_LogSessList_entry_pointer_physical_local_offset + 40 : l_LogSessList_entry_pointer_physical_local_offset + 44].encode('hex'))
username_pointer_base0, username_pointer_offset0, username_pointer_addr0 = self.convert(username_pointer[0])
username_pointer_base1, username_pointer_offset1, username_pointer_addr1 = self.convert(username_pointer[1])
username_pointer_base2, username_pointer_offset2, username_pointer_addr2 = self.convert(username_pointer[2])
```

Figure 26 Routine to find the address which has the account name

Then you need to verify whether the account name corresponds to the saved value in the routine as shown in [Figure 27].

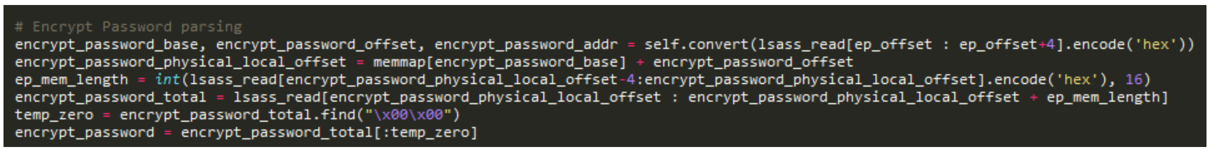


```
ep_offset = None
username = ""
try:
    username_offset = memmap[username_pointer_base0] + username_pointer_offset0
    if (lsass_read[username_offset : username_offset + 1] >= 'a' and lsass_read[username_offset : username_offset + 1] <= 'z') or \
    (lsass_read[username_offset : username_offset + 1] >= 'A' and lsass_read[username_offset : username_offset + 1] <= 'Z'):
        ep_offset = l_LogSessList_entry_pointer_physical_local_offset+48
        username = lsass_read[username_offset : username_offset + 16]
except:
    pass

try:
    username_offset = memmap[username_pointer_base1] + username_pointer_offset1
    if (lsass_read[username_offset : username_offset + 1] >= 'a' and lsass_read[username_offset : username_offset + 1] <= 'z') or \
    (lsass_read[username_offset : username_offset + 1] >= 'A' and lsass_read[username_offset : username_offset + 1] <= 'Z'):
        ep_offset = l_LogSessList_entry_pointer_physical_local_offset+52
        username = lsass_read[username_offset : username_offset + 16]
except:
    pass

try:
    username_offset = memmap[username_pointer_base1] + username_pointer_offset2
    if (lsass_read[username_offset : username_offset + 1] >= 'a' and lsass_read[username_offset : username_offset + 1] <= 'z') or \
    (lsass_read[username_offset : username_offset + 1] >= 'A' and lsass_read[username_offset : username_offset + 1] <= 'Z'):
        ep_offset = l_LogSessList_entry_pointer_physical_local_offset+56
        username = lsass_read[username_offset : username_offset + 16]
except:
    pass
```

Figure 27 Routine to verify the account name

5.3. Extracting the encrypted password

There exists an encrypted password of relevant account in +0x10 away from the field which has the account name. Therefore when verifying the account name, if you find any valid account, you have to calculate the field address +0x10, where the account was found. Then you can extract and check the address value which has an encrypted password. As we mentioned in [Figure 17], the length of the character string of encrypted password is different from that of the encrypted password saved 4 byte before. So you have to do the extracting only by 0x00(NULL) in the relevant routine, and save the necessary value for the actual decryption. [Figure 28] shows the routine of extracting encrypted password.

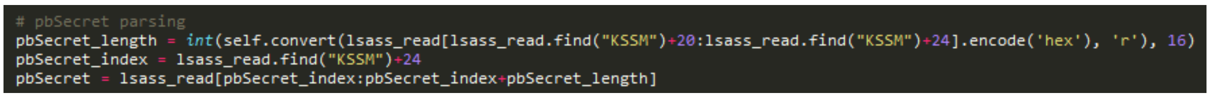


```
# Encrypt Password parsing
encrypt_password_base, encrypt_password_offset, encrypt_password_addr = self.convert(lsass_read[ep_offset : ep_offset+4].encode('hex'))
encrypt_password_physical_local_offset = memmap[encrypt_password_base] + encrypt_password_offset
ep_mem_length = int(lsass_read[encrypt_password_physical_local_offset-4:encrypt_password_physical_local_offset].encode('hex'), 16)
encrypt_password_total = lsass_read[encrypt_password_physical_local_offset : encrypt_password_physical_local_offset + ep_mem_length]
temp_zero = encrypt_password_total.find("\x00\x00")
encrypt_password = encrypt_password_total[:temp_zero]
```

Figure 28 Routine to extract the encrypted password

5.4. Extracting the value of pbSecret

The pbSecret actually plays the key role during the process of decryption. You need to find pbSecret space with specific signature called "KSSM", because the method you used in decompiling cannot be applied to find relevant value. The pbSecret has the length value of character string 16 byte after KSSM signature, and there exists a value of pbSecret right after it. [Figure 29] shows the routine of extracting the pbSecret.



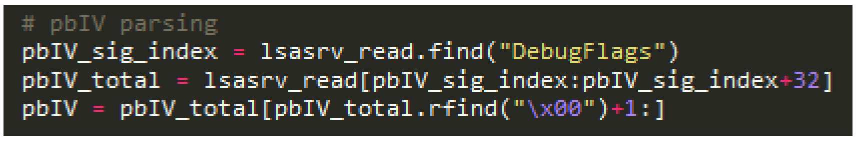
```
# pbSecret parsing
pbSecret_length = int(self.convert(lsass_read[lsass_read.find("KSSM")+20:lsass_read.find("KSSM")+24].encode('hex'), 'r'), 16)
pbSecret_index = lsass_read.find("KSSM")+24
pbSecret = lsass_read[pbSecret_index:pbSecret_index+pbSecret_length]
```

Figure 29 Routine that extract the pbSecret

5.5. Extracting the value of pbIV

You have to find the pbIV value through a specific signature as well as pbSecret. "DebugFlags" is recommended signature, and there is a value of 0x00 (NULL) between signature and pbIV value.

The size information of pbIV is not saved, but you can extract the size of 16 byte or the value from relevant value to 0x00 (NULL). [Figure 30] shows the routine of extracting the pbIV.



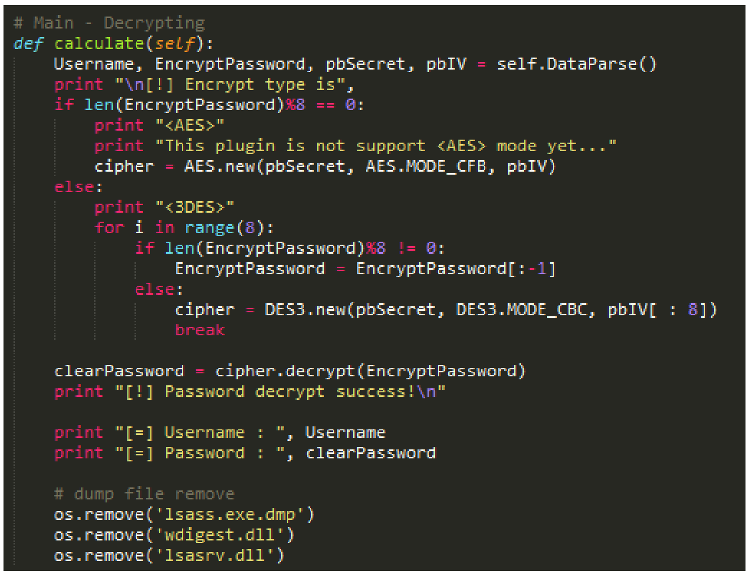
```
# pbIV parsing
pbIV_sig_index = lsasrv_read.find("DebugFlags")
pbIV_total = lsasrv_read[pbIV_sig_index:pbIV_sig_index+32]
pbIV = pbIV_total[pbIV_total.rfind("\x00")+1:]
```

Figure 30 Routine that extract the pbIV

.

5.6. Decrypting the password

When decrypting the password, you can use the Crypto function provided by Python. Initial vector (pbIV), key (pbSecret), and encrypted password are necessary. When the decryption using the 3DES algorithm is done, it deletes the dumps and every image files already used. [Figure 31] shows the routine of password decryption.

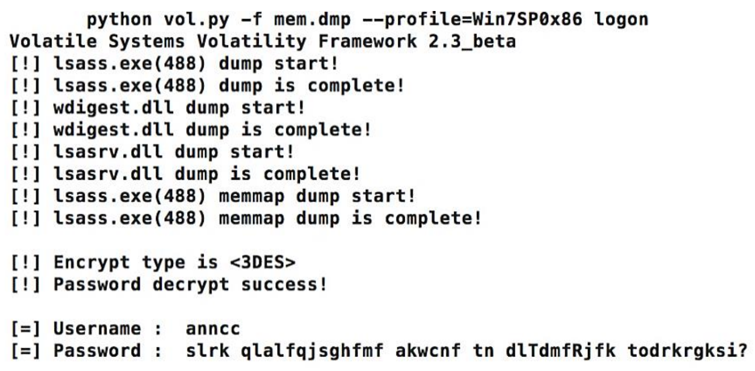


```
# Main - Decrypting
def calculate(self):
    Username, EncryptPassword, pbSecret, pbIV = self.DataParse()
    print "\n[!] Encrypt type is",
    if len(EncryptPassword)%8 == 0:
        print "<AES>"
        print "This plugin is not support <AES> mode yet..."
        cipher = AES.new(pbSecret, AES.MODE_CFB, pbIV)
    else:
        print "<3DES>"
        for i in range(8):
            if len(EncryptPassword)%8 != 0:
                EncryptPassword = EncryptPassword[:-1]
            else:
                cipher = DES3.new(pbSecret, DES3.MODE_CBC, pbIV[ : 8])
                break

    clearPassword = cipher.decrypt(EncryptPassword)
    print "[!] Password decrypt success!\n"

    print "[=] Username : ", Username
    print "[=] Password : ", clearPassword

    # dump file remove
    os.remove('lsass.exe.dmp')
    os.remove('wdigest.dll')
    os.remove('lsasrv.dll')
```

Figure 31 Decryption Routine

5.7. Plugin Result

[Figure 32] shows the result of executed relevant plugin.



```
       python vol.py -f mem.dmp --profile=Win7SP0x86 logon
Volatile Systems Volatility Framework 2.3_beta
[!] lsass.exe(488) dump start!
[!] lsass.exe(488) dump is complete!
[!] wdigest.dll dump start!
[!] wdigest.dll dump is complete!
[!] lsasrv.dll dump start!
[!] lsasrv.dll dump is complete!
[!] lsass.exe(488) memmap dump start!
[!] lsass.exe(488) memmap dump is complete!

[!] Encrypt type is <3DES>
[!] Password decrypt success!

[=] Username :  anncc
[=] Password :  slrk qlalfqjsghfmf akwcnf tn dlTdmfRjfk todrkrgksi?
```

Figure 32 The result of executed plugin

You can download the plugin from the following address.

https://gitlab.kr/For-MD/volatility-plugin-logon/blob/master/logon.py

6. Conclusion

First of all, we will keep upgrading this tool and show you a new method according to the computing environment which is changing. The environment which uses 32 bit system is now turning into an environment which uses 64 bit system, and the application range of this plugin is 50:50. Therefore we will apply the method of extracting the Windows account information in 64 bit system, and also add a new function to extract the account information from various authentication packages to broaden the application range of plugin. Finally, it is a big question for us how to extract the authentication session, the Windows account information when authenticating the domain, and multiple users within the memory considering the Active Directory environment.

In this article, we covered how to extract the information of Windows account from memory image in the view of digital forensics. In terms of digital forensics, the former way using dll injection is not appropriate, because it is against the integrity of memory. However, by the method we introduced in this article, you can extract the information of Windows account only by using the memory image on offline. Therefore we presume that it can be helpfully used in the field of investigation or security incidents.