배포 가능, 수정 불가

Windows Logon Password

Get Windows Logon Password using Wdigest in Memory Dump

2013-10-21

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

잘못은 그것을 통하여 우리가 발전할 수 있는 훈련이다. By 윌리엄 엘러리 채닝

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

1. 서론

기존에 존재하는 사용자의 윈도우 로그온 패스워드를 획득 하는 방법은 레지스트리와 윈도우 로그온 세션을 통해 NTLM 해시 값을 획득하고, 이를 크랙하는 방법이 주를 이루었다. 아래 [표 1]은 현재까지 잘 알려져 있는 사용자의 윈도우 로그온 패스워드 NTLM 해시 값을 획득하는 방법들이다. 조금 더 정확한 정보를 알고 싶다면 http://bernardodamele.blogspot.kr/ 의 "Dump Windows password hashes efficiently" 연재를 보도록 하자.

표 1 사용하는 파일 별 사용자의 윈도우 로그온 패스워드 NTLM 해시 획득 방법

사용 파일	패스워드의 NTLM 해시 획득 방법
SAM	SAM 하이브 파일의 값 복호화
NTDS.DIT	NTDS.DIT 의 데이터베이스 테이블 추출 후 복호화
NTDS.DIT /SAM	NTDS.DIT/SAM 하이브 파일의 Password History 정보를 이용
SECURITY	SECURITY 하이브 파일의 LSA Secret 복호화
SECURITY	SECURITY 하이브 파일의 Cached Domain Logon 정보를 이용
MSV1.0	윈도우 로그온 세션의 Credential 정보를 이용

해당 기법을 통해 획득하는 정보는 모두 NTLM 해시로 되어 있으며, 패스워드 크랙 도구를 통해 크랙 해야 하는 문제점이 있다. 이는 패스워드가 크랙을 할 수 없을 정도로 많은 시간을 소요하도록 길게 설정되어 있다면, 사용자의 윈도우 로그온 패스워드를 획득할 수 없는 문제점이 존재한다. 그러나 이와 같은 문제점을 해결하기 위해 라이브 상태에서 DLL Injection 을 하여, 아무리 긴 패스워드라 할지라도 사용자의 윈도우 로그온 패스워드를 평문으로 출력하는 도구 "Mimikatz"가 2012 년도에 발표[1]되었다.

본 문서에서는 해당 도구가 사용한 방법 중 하나인 "Wdigest 를 이용한 사용자의 윈도우 로그온 패스워드 추출" 방법을 메모리 덤프에 적용하여, 메모리 포렌식 시에 수사관들에게 조금이나마 도움을 주고자 한다.

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

2. 윈도우 인증 패키지

윈도우 인증 패키지란 윈도우 보안을 구현하는 주요 구성 요소 중 하나로, Lsass 프로세스 컨텍스트와 클라이언트 프로세스 내에서 실행되는 DLL 들을 포함한다. 이 중 인증 DLL 의 역할은 주어진 사용자 이름과 패스워드가 일치하는지 여부를 검사하는 것이며, 인증 정보가 일치하는 경우 Lsass 에 좀 더 상세한 사용자의 정보를 반환하고 이를 통해 Lsass 가 토큰을 생성한다. 대표적인 윈도우 인증 패키지는 MSV1_0, TsPkg, Wdigest, LiveSSP, Kerberos, SSP 등이 있으며, 각각 패키지 별로 Remote RDP, 웹 서비스 등 다양한 용도에 의해 구현되었다. 이는 Challenge-Response 방식을 위해 특정 필요 데이터를 메모리에 항상 가지고 있는 특징이 있다. 본 문서에서는 윈도우 인증 패키지 중 Wdigest 에 대해서만 알아보도록 한다.

2.1 WDigest.dll

Wdigest.dll 은 윈도우 XP 시스템에서 처음 소개 되었으며, HTTP 다이제스트 인증 및 SASL (Simple Authentication Security Layer) 교환에서 사용자를 인증하기 위해 개발되었다. 이는 다이제스트 인증에 사용되어 NTLM 프로토콜과 같이 Challenge-Response 방식을 사용하며, MD5 해시 또는 메시지 다이제스트로 네트워크를 통해 인증서를 전송하고, 기본 인증보다 향상된 보안을 제공한다. 그러나 인증을 위한 키를 도출하기 위해서는 사용자의 평문 패스워드가 필요한 특징이 있어, 이를 악용 가능하다. 아래 [그림 1]과 [표 2]는 다이제스트 인증 아키텍처와 각각의 요소에 대한 설명이다.

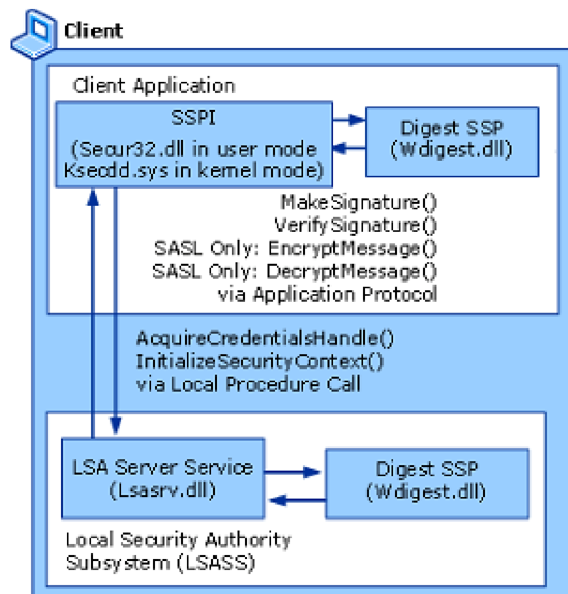

그림 1 다이제스트 인증 아키텍처

표 2 다이제스트 인증 요소

파일	설명
Wdigest.dll	LDAP 및 웹 인증을 위해 사용되는 SSP 를 구현
Lsasrv.dll	LSA 의 보안 서비스 관리(보안 정책 및 동작)
Secur32.dll	유저 모드 응용프로그램 SSPI 를 구현
Ksecdd.sys	커널 보안 장치 드라이버가 유저 모드에서 Lsass 와 통신하는데 사용

3. 라이브 상태에서 윈도우 로그온 패스워드 추출

라이브 상태에서 DLL Injection을 이용해 Wdigest에서 윈도우 로그온 패스워드를 추출하는 과정은 아래 [그림 2]와 같다.

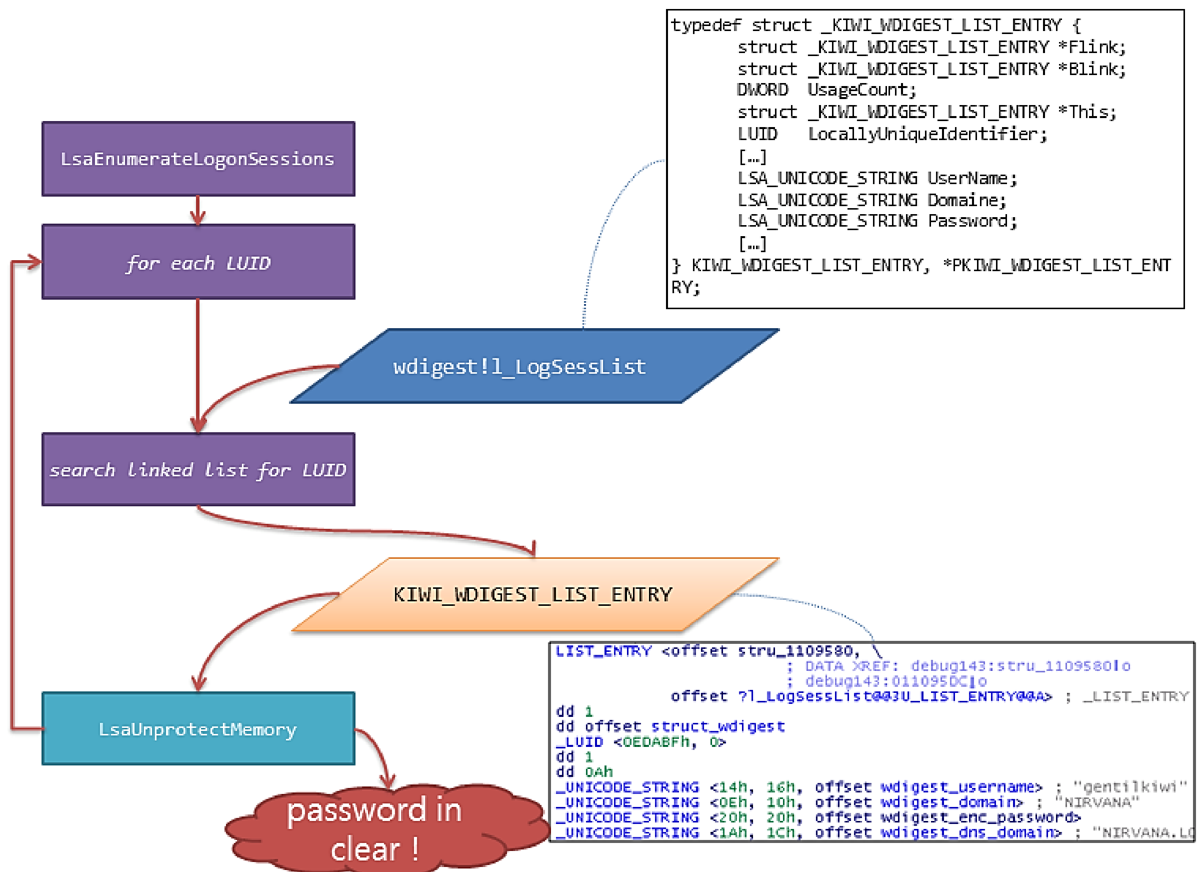

그림 2 라이브 상태에서 WDigest를 이용한 윈도우 로그온 패스워드 추출 과정

추출 과정의 수행되는 함수/dll 요소 별 동작 순서는 아래와 같다.

1. 우선 Lsass의 LsaEnumerateLogonSessions 함수를 통해 시스템에 존재하는 로그온 세션 식별자들(LUIDs)과 세션들의 수를 획득한다. 아래 [그림 3]은 LsaEnumerateLogonSessions 함수[2]이다.

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

그림 3 LsaEnumerateLogonSessions 함수

LogonSessionCount 포인터 변수는 로그온 세션의 개수를 가지고, LogonSessionList 포인터 변수는 로그온 세션 식별자들 중 첫 번째 요소의 주소 값을 가진다. 이를 통하여 시스템에 존재하는 로그온 세션 리스트를 추적 가능하다.

2. WDigest.dll의 l_LogSessList는 LIST_ENTRY 구조체로 되어 있으며, Flink, Blink, LUID를 비롯해 유니코드 문자열로 사용자 명, 도메인, 암호화된 패스워드, 도메인 DNS 등을 가지고 있다.

3. 이후 Lsasrv.dll의 LsaUnprotectedMemory 함수를 통해 l_LogSessList에서 획득한 암호화된 패스워드를 복호화 가능하다. 아래 [그림 4]는 LsaUnprotectedMemory 함수[3]이다.

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

그림 4 LsaUnprotectedMemory 함수

---

[2] http://msdn.microsoft.com/en-us/library/windows/desktop/aa378275(v=vs.85).aspx

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

4. LsaUnProtectedMemory 함수를 디컴파일 하면 아래 [그림 5]와 같다.

```
1 void __stdcall LsaUnprotectMemory(PUCHAR pbOutput, ULONG cbOutput)
2 {
3   LsaEncryptMemory(pbOutput, cbOutput, 0);
4 }
```

[그림 5] LsaUnprotectMemory 함수 디컴파일

내부적으로 LsaEncryptMemory 함수를 호출하며, 아래 [그림 6]은 LsaEncryptMemory 함수의 디컴파일 결과 이다.

```
1 NTSTATUS __stdcall LsaEncryptMemory(NTSTATUS pbOutput, ULONG cbOutput, _DWORD Mode)
2 {
3   NTSTATUS result; // eax@1
4   ULONG pcbResult; // [sp+0h] [bp-1Ch]@1
5   ULONG cbIV; // [sp+4h] [bp-18h]@1
6   int pbIV; // [sp+8h] [bp-14h]@3
7   _DWORD v7; // [sp+Ch] [bp-10h]@3
8   _DWORD v8; // [sp+10h] [bp-Ch]@3
9   _DWORD v9; // [sp+14h] [bp-8h]@3
10
11  result = pbOutput;
12  cbIV = 8;
13  pcbResult = 0;
14  if ( pbOutput && cbOutput )
15  {
16    pbIV = InitializationVector[0];
17    v7 = InitializationVector[1];
18    v8 = InitializationVector[2];
19    v9 = InitializationVector[3];
20    JUMPOUT((cbOutput & 7) != 0, byte_75BDC8A5);
21    if ( Mode )
22    {
23      if ( Mode == 1 )
24        result = BCryptEncrypt(h3DesKey, pbOutput, cbOutput, 0, &pbIV, cbIV, pbOutput, cbOutput, &pcbResult, 0);
25    }
26    else
27    {
28      result = BCryptDecrypt(h3DesKey, pbOutput, cbOutput, 0, &pbIV, cbIV, pbOutput, cbOutput, &pcbResult, 0);
29    }
30  }
31  return result;
32 }
```

그림 6 LsaEncryptMemory 함수 디컴파일

위 디컴파일 결과를 통해 결론적으로 Lsasrv.dll의 LsaEncryptMemory 함수가 3DesKey 해들값과 pbIV를 이용하여 암호화된 패스워드 복호화를 수행하는 것을 알 수 있다.

지금까지 전체적인 동작 과정을 설명하였다. 해당 동작 과정을 코드를 통해 살펴보고 싶은 분은 https://github.com/thomhastings/mimikatz-en/ 에서 확인해 보도록 하자.

4. 메모리 덤프에서 윈도우 로그온 정보 추출

추출을 수행하기 전에 메모리 덤프에서 WDigest를 이용해 윈도우 로그온 패스워드를 획득하기 위해 알아야 할 부분부터 정리해보도록 하겠다. 아래 [표 3]을 보자.

표 3 알아야 할 부분

분류	설명
필요한 dll	WDigest.dll : 암호화된 패스워드 값의 주소를 가지고 있음 Lsasrv.dll : 암호화된 패스워드의 복호화를 위해 필요
찾아야 할 값	WDigest.dll : l_LogSessList 의 암호화된 패스워드 값의 주소 Lsasrv.dll : 3DesKey 의 Handle 값, pbIV 값

위 [표 3]에서 언급한 내용만 보면 패스워드 추출은 단순해 보일지도 모른다. 그러나 메모리 덤프에서 해당 값을 찾고 이를 추적하는 것은 꽤나 복잡한 순서를 거쳐야 한다. 이는 우리가 가지고 있는 메모리 덤프 파일 자체로는 물리 주소로만 접근 할 수 있고, dll이 가지고 있는 포인터 값은 가상 주소를 기준으로 되어 있기 때문이다.

이제부터 본격적으로 메모리 덤프에서 윈도우 로그온 패스워드를 추출하는 방법을 살펴보도록 하겠다. 우선 WDigest.dll과 Lsasrv.dll을 추출하기 위해 상위 프로세스인 Lsass.exe의 PID를 확인하자. 아래 [그림 7]은 Volatility의 pslist 플러그인을 활용하여 Lsass.exe의 PID를 확인한 화면이다.

```
      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
```

그림 7 Lsass.exe의 PID 확인

이후 Lsass.exe가 실행되고 있을 때 할당된 메모리 영역의 값들을 확인하기 위해 memdump 플러그인을 이용해 Lsass.exe의 메모리 덤프를 획득한다. 아래 [그림 8]은 PID 488번인 Lsass.exe의 메모리 덤프 명령 수행 화면이다.

```
        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
```

그림 8 Lsass.exe 메모리 덤프 수행

지금까지 과정을 통해 "전체 메모리 덤프 -> 추출해야 할 모든 값"이 있는 Lsass.exe 메모리 덤프로 윈도우 로그온 패스워드를 획득하기 위해 분석해야 할 덤프 파일 크기를 줄였다. 앞서 언급했듯이 Lsass.exe 메모리 덤프 역시 물리 주소로만 접근 가능하다. 그러므로 가상 주소와 물리 주소의 맵핑을 위한 메모리 맵 파일을 생성해야 한다. 아래 [그림 9]는 memmap 플러그인을 통하여 해당 메모리 덤프의 메모리 맵을 추출하는 화면이다.

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

그림 9 메모리 맵 수집

다음으로 윈도우 로그온 패스워드 추출에 필요한 dll 중 하나인 WDigest.dll을 덤프 한다. 아래 [그림 10]은 Wdigest.dll을 덤프 하는 명령 수행 화면이다.

```
        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
```

그림 10 WDigest.dll 덤프

WDigest.dll은 암호화된 패스워드 값의 주소를 가지고 있으며, 이를 추적하기 위한 리스트의 첫 시작주소는 l_LogSessList 에서 확인 가능하다. 이는 LIST_ENTRY 구조체로 되어 있다. 아래 [그림 11]은 l_LogSessList에 저장된 값을 나타낸다.

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

그림 11 l_LogSessList 확인

0x168D50는 사용자 로그온 세션 리스트 중 가장 첫 번째 요소의 가상 주소를 나타낸다. 해당

주소를 포함하는 영역을 찾아 Lsass.exe 메모리 덤프 파일의 물리 주소를 확인해 보자. 아래 [그림 12]는 메모리 맵 파일에서 0x168D50을 포함하는 가상 주소 영역을 검색하는 화면이다.

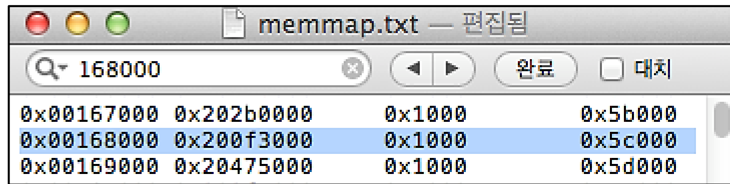

그림 12 0x168D50 포함 주소 검색

0x00168000부터 0x1000 사이즈의 영역이 Lsass.exe 메모리 덤프 파일의 0x5c000부터 매핑 되어 있다. Lsass.exe 물리 메모리 덤프에서 0x5C00 + (0x168D50 - 0x168D00) = 0x5CD50으로 이동하여 값을 확인해 보자. 아래 [그림 13]은 WinHex에서 오프셋 0x5CD50 지점을 확인한 화면이다.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
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 ¸

그림 13 Lsass.exe 물리 메모리 덤프에서 0x5CD50 지점 확인

0x5CD50으로부터 0x20 지점은 사용자 계정이 존재하는 곳이며, 내용이 없을 경우 바로 다음 4 바이트를 확인한다. 이 메모리 덤프에서는 오프셋 0x5CD74부터 4바이트에 있는 값인 0x001C80A8이 사용자 계정이 존재하는 영역이며, 아래 [그림 14]는 이를 확인한 화면이다.

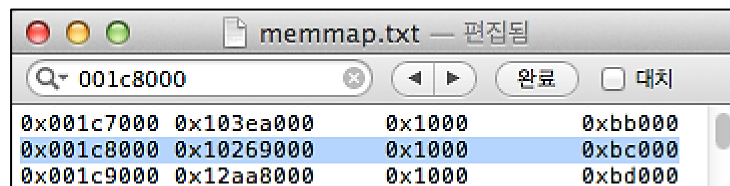

그림 14 0x001C80A8 포함 주소 검색

Lsass.exe 메모리 덤프의 0xBC000 + (0x1C80A8 - 0x1C8000) = 0xBC0A8 지점으로 이동하여 보자. 유니코드로 사용자 계정이 존재하는 것을 확인할 수 있다. 아래 [그림 15]는 사용자 계정을 확인한 화면이다.

Offset	0–7	8–F	ASCII
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 ▌

그림 15 사용자 계정 확인

암호화된 사용자의 윈도우 로그온 패스워드는 사용자 계정 정보 주소를 가지는 0x5CD74 + 0x10 오프셋에 존재한다. [그림 13]의 0x5CD84 지점의 값 0x001A1DB8의 물리 주소를 아래 [그림 16]과 같이 확인해 보자.

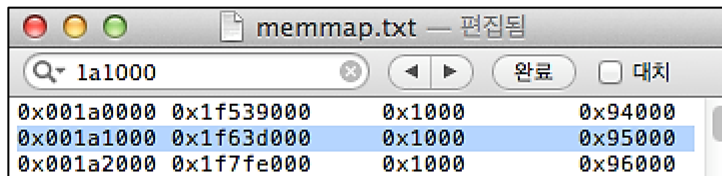

그림 16 0x001A1DB8 포함 주소 검색

Lsass.exe 메모리 덤프의 0x95DB8 지점으로 가면 아래 [그림 17]과 헥사 값이 존재한다. 해당 부분이 바로 WDigest.dll에서 획득 가능한 암호화된 사용자의 패스워드 이다. 중간에 0x00(NULL)이 포함되어 실제로는 0x95E23 까지가 암호화된 사용자의 윈도우 로그온 패스워드의 유효 값 임을 알 수 있다.

Offset	0–7	8–F	ASCII
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

그림 17 암호화된 사용자의 윈도우 로그온 패스워드 확인

다음으로 암호화된 사용자의 윈도우 로그온 패스워드를 복호화 하기 위해 필요한 dll인 Lsasrv.dll을 덤프 한다. 아래 [그림 18]은 해당 dll 덤프 화면이다. 플러그인 명령 방식은 기존 WDigest.dll 덤프와 동일하다.

```
      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
```

그림 18 Lsasrv.dll 덤프

3DesKey 핸들 값은 LsaEncryptMemory 함수의 패스워드 복호화 과정에 사용되며, 아래 [그림 19]와 같이 알 수 있다. 그러나 실제로 우리가 필요한 값은 이 3DesKey 핸들값이 아닌 pbSecret 값이다. 복호화를 수행하는 최종 값은 pbSecret와 pbIV, 암호화된 사용자의 윈도우 로그온 패스워드 이 3가지로 이루어지기 때문이다. pbSecret 값은 3DesKey 주소 + 0x3C 위치에 존재한다.

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

그림 19 3DesKey 주소 확인

0x31000은 물리주소 0xFB000에 맵핑 되는 것을 아래 [그림 20]을 통해 알 수 있다.

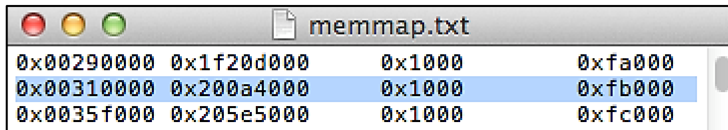

그림 20 0x310000 포함 주소 검색

pbSecret는 위에서 언급했듯이 0x3C 위치 이후에 존재하므로, Lsass.exe 메모리 덤프의 0xFB000 + 0x3C = 0xFB03C로 이동해보자. 아래 [그림 21]과 같이 앞에 4바이트의 크기가 존재하고, 뒤에 헥사 값이 존재하는 것을 확인할 수 있다.

```
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
```

그림 21 pbSecret 값 확인

마지막으로 pbIV 값은 단순히 Lsasrv.dll의 InitializationVector[4] 배열의 첫 번째 인덱스 값을 확인하면 알 수 있다. 아래 [그림 22]는 pbIV 값을 확인한 화면이다.

```
.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
```

그림 22 pbIV 값 확인

마지막으로 획득한 값을 통해 Python으로 3Des 를 구현하여 실제 사용자 로그온 패스워드를 확인해 보겠다. 아래 [그림 23]은 해당 Python 코드이다.

```
1 from Crypto.Cipher import DES3
2
3 pbSecret = "\x66\x45\x06\xAC\x9C\x29\xAE\xFF\x48\x6D\x7D\x19\x80\x37\x81\xD7\x76\x9C\xB5\x74\x90\x22\x4B\xA3"
4 pbIV = "\xE1\x82\x28\xA3\x59\x45\x20\x3F\x19\xE4\x81\x41\xDB\x87\xC5\x6B"
5 enc_value = "\x77\x1A\x61\xA1\xDB\xBA\xAC\x9F\xC2\x84\x8D\x60\x29\x9B\xA4\x7E\x61\xE0\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\xFE\x86\xDF\xD8\x1B\x00\xDB\x45\x95\x9F\xE3\xD4\xEB\x74\x64\x36\x50\x67\xD2\x78\x8D\xC1\x4B\x8B\xDF"
6
7 cipher = DES3.new(pbSecret, DES3.MODE_CBC, pbIV[ : 8])
8 password = cipher.decrypt(enc_value)
9 print "Password is \n[\t%s\t]"%password
```

그림 23 사용자 암호 복호화를 위한 파이썬 코드

[그림 24]는 코드 수행 결과이다. 패스워드 길이가 매우 길더라도 복호화가 가능하다.

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

그림 24 복호화 코드 수행 결과

5. Volatility Plugin - logon

본 문서에서 소개한 메모리에서 윈도우 사용자 로그온 패스워드를 추출하는 방법을 토대로 Volatility 플러그인을 제작하였다. 이번 절에서는 간단하게 해당 플러그인의 동작을 설명하고자 한다. 대략적인 플러그인의 동작 순서를 살펴보도록 하자.

5.1. Lsass.exe, Wdigest.dll, Lsasrv.dll 메모리 덤프

가장 먼저 필요한 정보들을 추출하기 위한 영역들을 추출하는 작업을 수행한다. 본 플러그인에서는 필요 영역을 소유하고 있는 프로세스의 메모리 영역을 덤프 한다. 아래 [그림 25]는 덤프를 수행하는 함수이다.

```
# 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)
```

그림 25 dump() 함수

각각의 필요 이미지들은 Volatility에서 제공하는 기본적인 덤프 모듈들을 이용 해 덤프하며, 이때 덤프 목록에서 필요한 이미지들의 이름을 이용 해 필터 하여 자동으로 덤프 하여 준다.

5.2. 사용자 계정 명 추출

메모리에서 사용자 계정 명을 추출 할 때에는 l_LogSessList+0x20의 값(사용자 계정 명을 유니코드로 가지고 있는 주소)를 확인하고, 만약 주소가 유효하지 않다면 다음 4 바이트 필드의 주소를 확인하여 계정을 찾는 방법을 사용한다. 그러나 플러그인에서는 어떤 필드에 어떤 주소가 계정을 가지고 있는지 쉽게 파악하기 힘들기 때문에, 특정 영역(4바이트 * 3회)의 주소 값을 확인 후, 해당 주소로 이동해 유효한 문자열이 존재하는지 점검하여 계정 존재 여부를 판단한다. 유효한 계정의 검증은 윈도우 계정 명 정책을 따른다. 아래 [그림 26]은 계정명이 저장되어 있는 주소를 찾는 루틴이다.

```
# 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])
```

그림 26 계정명이 저장되어 있는 주소 찾는 루틴

그 후 아래 [그림 27]과 같이 다음 루틴에서 해당 후보 군들의 저장 값이 계정 명이 맞는지 검증을 수행한다.

```
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
```

그림 27 계정명 검증 루틴

5.3. 암호화 된 패스워드 추출

계정 명이 저장되어 있는 주소를 가진 필드로부터 +0x10 영역에 해당 계정의 암호화된 패스워드가 존재한다. 그러므로 계정 명 검증 시 유효한 계정을 발견하면 해당 계정이 발견되었던 필드의 주소 +0x10 연산을 하여 암호화 된 패스워드가 저장된 주소 값을 확인 및 추출한다. 이때 [그림 17]에서도 언급하였지만 암호화 된 패스워드는 바로 앞 4바이트에 저장된 암호화 된 패스워드 문자열 길이와 다르다. 그러므로 해당 루틴에서 0x00(NULL) 까지만 추출하여, 실제 복호화에 필요한 값만 저장한다. 아래 [그림 28]은 암호화 된 패스워드 추출 루틴이다.

```
# 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]
```

그림 28 암호화 된 패스워드 추출 루틴

5.4. pbSecret 값 추출

pbSecret은 사실 상 복호화 과정에서 키 역할을 하는 값이다. 해당 값을 찾기 위해서는 디컴파일에서 수행한 방법을 동일하게 적용할 수 없기 때문에, "KSSM" 이라는 특정 시그니처로 pbSecret 영역을 찾아야 한다. pbSecret은 KSSM 시그니처 이후 16바이트 뒤에 문자열 길이 값을 두고 있고, 바로 다음부터 pbSecret 값을 가지고 있다. 아래 [그림 29]는 pbSecret 추출 루틴이다.

```
# 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]
```

그림 29 pbSecret 추출 루틴

5.5. pbIV 값 추출

pbIV 값 또한 pbSecret처럼 특정 시그니처를 통해 값을 찾아야 한다. 이때 "DebugFlags" 라는 시그니처를 사용할 수 있으며, 시그니처와 pbIV 값 사이에 0x00(NULL) 값이 있다. pbIV는 크기 정보는 저장되어 있지 않으나, 보통 16바이트 크기 또는 해당 값부터 0x00(NULL)까지 추출하면 된다. 아래 [그림 30]은 pbIV 추출 루틴이다.

```
# 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:]
```

그림 30 pbIV 추출 루틴

.

5.6. 패스워드 복호화

패스워드 복호화는 파이썬에서 제공하는 Crypto 함수로 수행한다. 초기벡터(pbIV), 키(pbSecret), 암호화 된 문자열(EncryptedPassword)이 필요하며, 3DES 알고리즘을 사용하여 복호화가 끝나면 덤프 및 사용 했던 이미지 파일들을 모두 지워준다. 아래 [그림 31]은 패스워드 복호화 루틴이다.

```
# 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')
```

그림 31 복호화 루틴

.

아래 [그림 32]는 해당 플러그인 수행 결과 이다.

```
      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?
```

그림 32 플러그인 수행 결과

해당 플러그인은 다음 주소에서 확인하고 다운로드 받을 수 있다.

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

6. 결론

먼저 해당 도구는 현재 바뀌고 있는 컴퓨팅 환경에 맞추어 계속해서 개발하고, 방법을 내놓을 것이다. 32bit 환경에서 64bit 환경으로 변하고 있는 현재, 해당 플러그인의 적용 범위는 50:50이다. 그러므로 64bit 환경에서의 윈도우 계정정보 추출 방법을 적용하고, 다양한 인증 패키지에서 계정정보를 추출하는 기능을 추가 적용하여 플러그인의 적용 범위를 넓혀야 한다고 생각한다. 마지막으로 Active Directory 환경도 고려하여 해당 환경에서 다중 사용자와 도메인 인증 때의 윈도우 계정정보와 인증세션을 어떻게 메모리 내에서 추출 할 것인지도 앞으로 해결해야 할 과제인 듯 하다.

해당 문서에서는 디지털 포렌식 관점에서 메모리 이미지에서 윈도우 계정정보를 추출하는 방법에 대해서 설명하였다. 기존에 공개되었던 dll Injection을 이용한 방법은 디지털 포렌식 관점에서 볼 때 메모리의 무결성을 파괴하므로 적절하지 못한 방법이다. 하지만, 이번에 소개한 방법은 오프라인 상에서 메모리 이미지 만을 가지고 윈도우 계정정보를 추출할 수 있어 추후 수사권, 또는 침해사고 등의 영역에서 유용하게 사용 될 수 있을 거라 생각 된다.