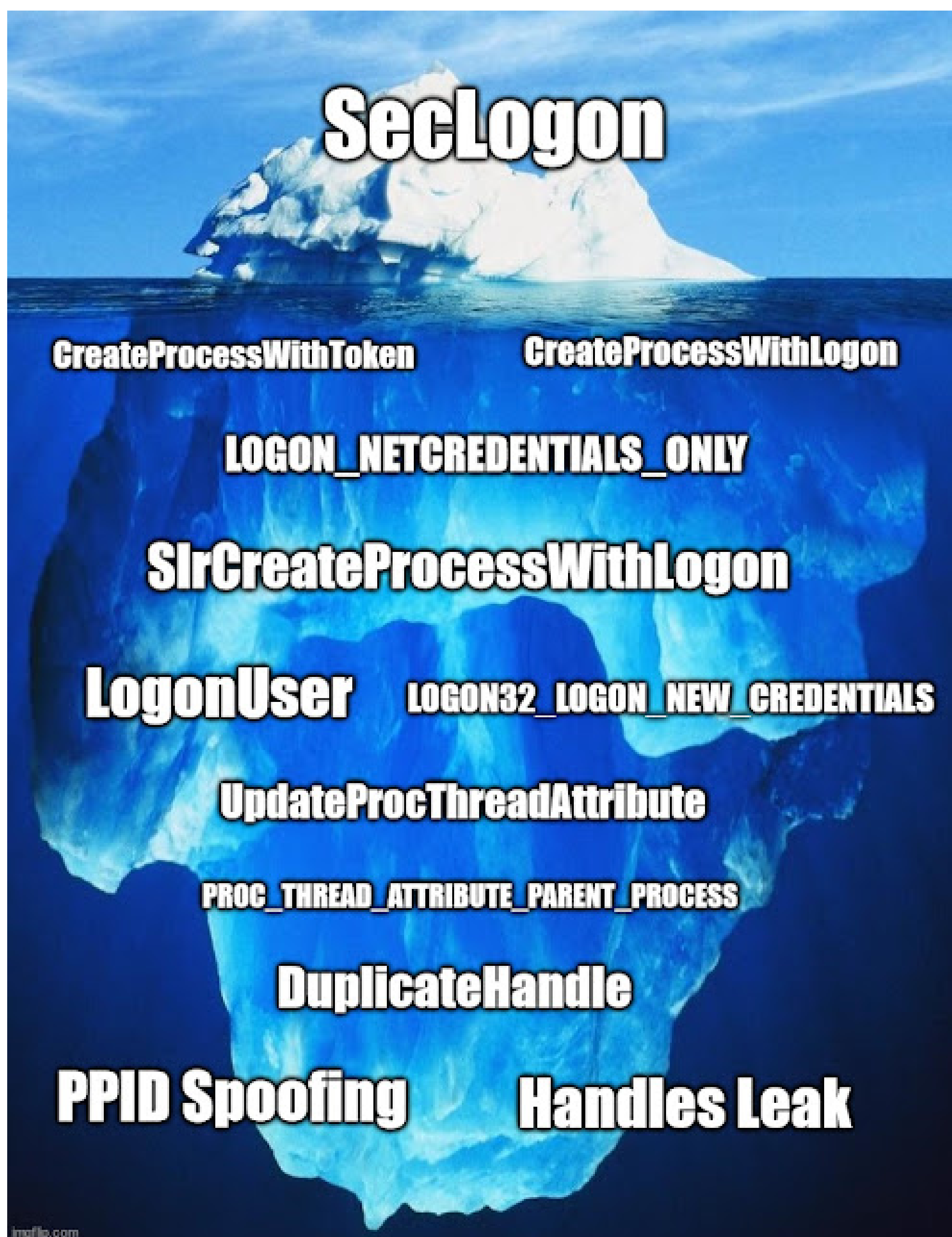
The hidden side of Seclogon part 2: Abusing leaked handles to dump LSASS memory

splintercod3.blogspot.com/p/the-hidden-side-of-seclogon-part-2.html

by splinter_code - 7 December 2021



Credential dumping is one of the most common techniques leveraged by attackers to compromise an infrastructure. It allows to steal sensitive credential information and enables attackers to further move **laterally** on the target environment.

On Windows systems there is a **SSO** (Single Sign-On) mechanism in which the user types the password only once and is automatically logged on every time it is needed as long as the user session on the operating system is alive. The main advantage of it is improving the **usability** of the system by not requesting credentials multiple times.

The main drawback of this approach is that those credentials must be **stored** somewhere. If the system has to authenticate the user automatically, the system must hold the user credentials in some form, that's a fact.

In particular, for Windows systems the process in charge for this is **lsass.exe** (Local Security Authority Subsystem Service).
Lsass is a critical process and contains a pile of treasures from an attacker perspective. For this reason **dumping** the memory of lsass process is something often performed when attackers carry out their malicious operations.

With that in mind most EDR/AV try to **protect** the process memory from **unauthorized/malicious access**. Like every evasion/detection method is always a game of cat-n-mouse.
New detection methods could be created only if attacking methods are known, that's another fact.

Considering that, in this blog post i'm going to release a new undocumented/unknown way to perform a memory dump of LSASS in a **stealthy** way. Giving detailed internals on how it works, it should ease (making known what is unknown) the development of effective detection.

On the edges of detection

The most **simple** way of dumping **lsass** memory usually involves two main operations:

- Opening a process handle to the **lsass PID** through an **OpenProcess** call with the access *PROCESS_QUERY_INFORMATION* and *PROCESS_VM_READ*;
- Using **MiniDumpWriteDump** to read all the process address space of lsass and save it into a file on the disk. Note that *MiniDumpWriteDump* heavily relies on the usage of the **NtReadVirtualMemory** system call that allows it to read memory of **remote** processes.

Now, where does the **detection** of lsass memory dumping usually occur? On both operations!

The first detection spot usually occurs on the usage of **OpenProcess**/**NtOpenProcess**.
The Windows kernel allows your driver to register a list of **callback routines** for thread, process, and desktop **handle** operations. This can be achieved through **ObRegisterCallbacks**.
Two structs are required to register a new callback: **OB_CALLBACK_REGISTRATION** and **OB_OPERATION_REGISTRATION**.

In particular, the **OB_OPERATION_REGISTRATION** struct allows to specify a combination of parameters to **monitor** any newly **created/duplicate** process **handle** directly from the kernel.

To mention an example, **Sysmon** event id 10 is based on this mechanism.

The second detection spot usually occurs on the usage of **NtReadVirtualMemory**, used internally also by **ReadProcessMemory**.
In this case the implementation may vary.
The most used approach is the **Inline Hooking** to intercept *NtReadVirtualMemory* calls that target the lsass process. The problem with this approach is that the monitoring occurs at the same ring level of the process itself, so techniques like direct system calls or unhooking would easily **bypass** this kind of detection.

A better approach is using the Threat Intelligence ETW to receive notifications directly from the kernel on specific functions invocation. E.g. whenever a *NtReadVirtualMemory* is called, the kernel function *EtwTiLogReadWriteVm* will be used to track the usage and send the event to the registered consumers.
Most modern and effective EDR go for this way.

Known dumping methods

It's important to highlight how some of the known (and most stealthy) dumping methods have inspired me in one way or another:

- **Evading WinDefender ATP credential-theft: a hit after a hit-and-miss start**by @matteomalvica and @b4rtik:

  Create a snapshot of the process in order to perform indirect memory reads by using the snapshot handle. The snapshot handle is then used in the MiniDumpWriteDump call instead of using the target process handle directly.
- **Duping AV with handles** by @SkelSec:

Reuses already opened handles to the lsass process thus avoiding a direct OpenProcess call on lsass.

- **Dumping LSASS in memory undetected using MirrorDump** by @_EthicalChaos_:

    Load an arbitrary LSA plugin that performs a duplication of the lsass process handle from the lsass process into the dumping process. So the dumping process has a ready to use process handle to lsass without invoking OpenProcess.

The above descriptions are just a brief, I strongly recommend you to read the blog posts in order to have more insights.

A sharp eye could catch that every mentioned method above tries to play on the **edges of detection** to stay under the radar.

I want to mention a specific thing about the technique of reusing already opened lsass handles. While it's a very valid technique, it has the clear disadvantage that on most systems you won't easily find a **handle holder** that's not lsass itself. You can verify it with a simple handles enumerator tool:

```
C:\Users\splintercode\Desktop\MalSeclogon\x64\Release>HandlesEnumerator.exe 816
Found opened process handle 0x06cc to target PID 816 in process lsass.exe(PID:816)
Found opened process handle 0x0718 to target PID 816 in process lsass.exe(PID:816)
Found opened process handle 0x0724 to target PID 816 in process lsass.exe(PID:816)
Found opened process handle 0x0734 to target PID 816 in process lsass.exe(PID:816)
Found opened process handle 0x0e6c to target PID 816 in process lsass.exe(PID:816)
```

So it means that if you want to get one of those process handles in your process you still need to open a handle to lsass to duplicate it.

Wouldn't it be nice to **coerce** a Windows System Service (not lsass clearly) to open a handle for you? Let's find out...

The bug: a bad assumption in SecLogon

Now you are wondering: why are you going to talk about the Secondary Logon Service (**seclogon**) in a blog post about credential dumping?

Long story short: I have spent a lot of time reversing the seclogon service while developing my **RunasCs** tool and I found many interesting weirdnesses in it.
If you notice, this blog post is a 2nd part of a never written part 1. I hope one day I can find enough time to document all the internals involved in RunasCs, I swear it contains some crazy stuff.

Back to the point… The seclogon service is a RPC server that basically exposes 1 function: **SeclCreateProcessWithLogonW**.
Whenever you use CreateProcessWithTokenW or CreateProcessWithLogonW in your program you will land in the seclogon service.
The seclogon function that implements all the logic of the process creation with **alternate credentials** is *SlrCreateProcessWithLogon* called from *SeclCreateProcessWithLogonW*. It has the following definition:

```
DWORD SlrCreateProcessWithLogon(
      RPC_BINDING_HANDLE BindingHandle,
      PSECONDARYLOGONINFOW psli,
      LPPROCESS_INFORMATION ProcessInformationOutput)
```

If you are familiar with the usage of the *CreateProcessWithTokenW* and *CreateProcessWithLogonW* you might know that the newly created process is a child of the calling process. So in some way the seclogon service must know which is the **PID** of the process performing the RPC call. Below the reversed code which get a handle to the caller:

```
LastError = RpcImpersonateClient(BindingHandle);
if ( LastError )
  goto SetFlagAndClearVariablesForExit;
flagIsImpersonating = 1;
flagIsImpersonating2 = 1;
hCaller = OpenProcess(
            PROCESS_QUERY_INFORMATION|PROCESS_CREATE_PROCESS|PROCESS_DUP_HANDLE,
            FALSE,
            psli->dwProcessId);
if ( !hCaller )
  goto ReturnLastError;
```

So it basically **impersonates** the caller and then tries to open the calling process with the PID **psli->dwProcessId**. This part of the code is very important because the hCaller process handle is then used for a series of operations to create the newly requested process. To change the parent PID of the new process, the process attributes are updated in order to match it with the **caller** (**hCaller**):

```
AttributeListSize = 48i64;
if ( !InitializeProcThreadAttributeList(lpAttributeList, 1u, 0, &AttributeListSize)
  || !UpdateProcThreadAttribute(
        lpAttributeList,
        0,
        PROC_THREAD_ATTRIBUTE_PARENT_PROCESS,
        &hCaller,
        8ui64,
        FALSE,
        FALSE) )
```

But where does the **psli->dwProcessId** value come from?

After some time reversing *advapi32.dll* I noticed that value is provided as an input to the RPC call, so technically we can provide any value we want. Below a snippet of code reversed from *CreateProcessWithLogonCommonW* called internally by *CreateProcessWithLogonW:*

```
*(_QWORD *)&pSeclSli.ulLogonIdLowPart = 0i64;
pSeclSli.ulLogonFlags = dwLogonFlags;
pSeclSli.ulProcessId = GetCurrentProcessId();
pSeclSli.ulCreationFlags = creationFlags;
pSeclSli.hWinsta = 0i64;
pSeclSli.hDesk = 0i64;
pSeclSli.ssUsername.pwsz = (wchar_t *)lpUsername;
```

And then the invocation of the RPC call:

```
LastError = c_SeclCreateProcessWithLogonW(&pSeclSli, &pSeclSlri);
if ( LastError )
  goto LABEL_64;
LastError = pSeclSlri.ulErrorCode;
```

So the seclogon service makes the **bad assumption** that if the caller is able to open the PID provided as the input of the RPC call, it means it's the process itself performing the call.

Great! Now we know we can spoof the PID of the caller and we can have a **PPID spoofing** primitive. But... we still miss one thing: some valid user credentials.

Luckily enough the seclogon service provides the so-called **LOGON_NETCREDENTIALS_ONLY** feature that allows to get (almost) a copy of the caller process token valid locally on the machine without providing any valid credentials.
To add some swag points, it's still possible to use *CreateProcessWithLogonW* without generating the stub to perform the RPC call directly.
If you have a look at the above reversed code of *CreateProcessWithLogonCommonW* you can notice that the value of the *pSeclSli.ulProcessId* is taken by calling **GetCurrentProcessId**.

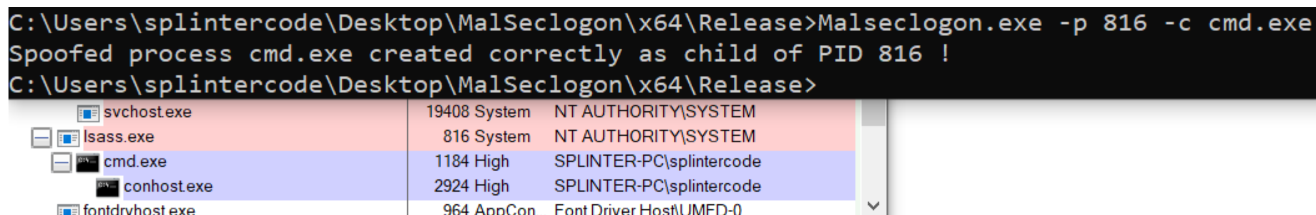
Below the reversed code:

```
DWORD __stdcall GetCurrentProcessId()
{
  return (DWORD)NtCurrentTeb()->ClientId.UniqueProcess;
}
```

It basically takes the PID from the current process **TEB**. So if we patch that value, we are sure that *CreateProcessWithLogonW* will the the job for us to **spoof** the pid while performing the RPC call:

```
void SpoofPidTeb(DWORD spoofedPid, PDWORD originalPid, PDWORD originalTid) {
   CLIENT_ID CSpoofedPid;
   DWORD oldProtection, oldProtection2;
   *originalPid = GetCurrentProcessId();
   *originalTid = GetCurrentThreadId();
   CLIENT_ID* pointerToTebPid = &(NtCurrentTeb()->ClientId);
   CSpoofedPid.UniqueProcess = (HANDLE)spoofedPid;
   CSpoofedPid.UniqueThread = (HANDLE)*originalTid;
   VirtualProtect(pointerToTebPid, sizeof(CLIENT_ID), PAGE_EXECUTE_READWRITE,
&oldProtection);
   memcpy(pointerToTebPid, &CSpoofedPid, sizeof(CLIENT_ID));
   VirtualProtect(pointerToTebPid, sizeof(CLIENT_ID), oldProtection, &oldProtection2);
}
```

Time for PPID spoofing **Demo**!



Unfortunately, even if the seclogon process opens a new process handle to lsass to create a child process, we **cannot duplicate** that handle from seclogon because it's **closed** shortly after. I didn't want to deal with race conditions, so I started to explore some alternative way to get my hands on a lsass process handle... (Well, technically it's possible to steal that lsass handle in a reliable way. But this is something for another blog post :D)

When a series of faults stack nicely together

At this point we have a nice PPID spoofing feature offered by the seclogon service, but still far away from having a lsass handle somewhere.

The first thing that came to my mind was **this vulnerability** discovered by @tiraniddo, and described here --> **Exploiting a Leaked Thread Handle**.
It's one of my favourite logic bugs I have ever seen, I strongly recommend you to go through that blog post.

This vulnerability allows you to **escalate** your privileges from a normal user to SYSTEM. What he had found out is that you can trick the seclogon service into **leaking a thread handle** of a thread running as **SYSTEM**. In particular, the leakage occurs through the usage of the **pseudo handle -2** (**GetCurrentThread**) provided as the **standard stream handle value** in the **startup information** structure and uses *CreateProcessWithLogonW* to **trigger** the seclogon service.

While the vulnerability has been fixed (and you cannot specify a pseudo handle anymore), it's still possible to **leak handles** from other processes if you **combine** it with the **PPID spoofing** primitive. You still need to have the rights to open the target process, so we are not crossing any security boundaries. But... to dump lsass memory you need admin privileges anyway so that's enough for our purpose :D

The scenario is a bit articulated, let's proceed step-by-step to understand the whole picture...

Either *CreateProcessWithLogonW* and *CreateProcessWithTokenW* allows to specify a **LPSTARTUPINFOW** struct for the parameters of the process. One of the things you can specify are the **Standard Streams** for the process that allows to redirect input and output of **console processes** on a **different stream**, e.g. a named pipe.
While these handles are inherited normally while performing a normal process creation, this won't happen in the case of the seclogon service. This occurs because the seclogon is not the real parent of the process so **handle inheritance** won't work by **design**.

For this reason the seclogon service has to "emulate" the same behavior.
Let's reverse how this is implemented...

When you specify the flag *STARTF_USESTDHANDLES* in the startup information a new flag is set to true, reversed code from *SlrCreateProcessWithLogon* below:

```
if ( (psli->lpStartupInfo->dwFlags & STARTF_USESTDHANDLES) != 0 )
{
  stdHandlesSpecified = TRUE;
}
else
{
  psli->lpStartupInfo->hStdInput = INVALID_HANDLE_VALUE;
  psli->lpStartupInfo->hStdOutput = INVALID_HANDLE_VALUE;
  psli->lpStartupInfo->hStdError = INVALID_HANDLE_VALUE;
}
```

Then the new process with the alternate credentials is created **suspended**:

```
if ( unicodeEnvironmentCreated )
  dwCreationFlags = EXTENDED_STARTUPINFO_PRESENT|CREATE_UNICODE_ENVIRONMENT|CREATE_SUSPENDED;
else
  dwCreationFlags = EXTENDED_STARTUPINFO_PRESENT|CREATE_SUSPENDED;
if ( CreateProcessAsUserW(
       hToken,
       psli->lpApplicationName,
       psli->lpCommandLine,
       &defaultSecurityAttributes,
       &defaultSecurityAttributes,
       FALSE,
       dwCreationFlags | psli->dwCreationFlags,
       psli->lpEnvironment,
       psli->lpCurrentDirectory,
       psli->lpStartupInfo,
       ProcessInformationOutput) )
{
```

Shortly after the **flag** signaling that the standard stream handles have been specified is **checked** and *SlpSetStdHandles* is invoked:

```
if ( stdHandlesSpecified )
{
  LOBYTE(resultSetStdHandles) = SlpSetStdHandles(
                                  ProcessInformationOutput->hProcess,
                                  hCaller,
                                  psli->lpStartupInfo->hStdInput,
                                  psli->lpStartupInfo->hStdOutput,
                                  psli->lpStartupInfo->hStdError);
```

So the function **SlpSetStdHandles** is the one in charge to **duplicate** the **standard stream handles** from the caller to the new process. For the bravest: here you can find the whole reversed function --
> https://gist.github.com/antonioCoco/706760df95749974b89546fb8d9fa445

In particular, this is the relevant snippet that allows to **leak handles**:

```
for ( PebPtrStdHandleTargetProcessX64 = &PebPtrStdInputX64; ; PebPtrStdHandleTargetProcessX64 += 3 )
{
  hStdHandleSourceProcess = *(PebPtrStdHandleTargetProcessX64 - 1);// based on for iteration = hStdInputStack, hStdOutputStack, hStdErrorStack
  if ( hStdHandleSourceProcess )
  {
    if ( (*(PebPtrStdHandleTargetProcessX64 - 1) & 0x10000003) != 3 )// *(PebPtrStdHandleTargetProcessX64 - 1) = hStdHandleSourceProcess
    {
      if ( !DuplicateHandle(
              hCaller,
              hStdHandleSourceProcess,
              hNewProcess,
              &hDupStdHandle,
              0,
              TRUE,
              DUPLICATE_SAME_ACCESS) )
```

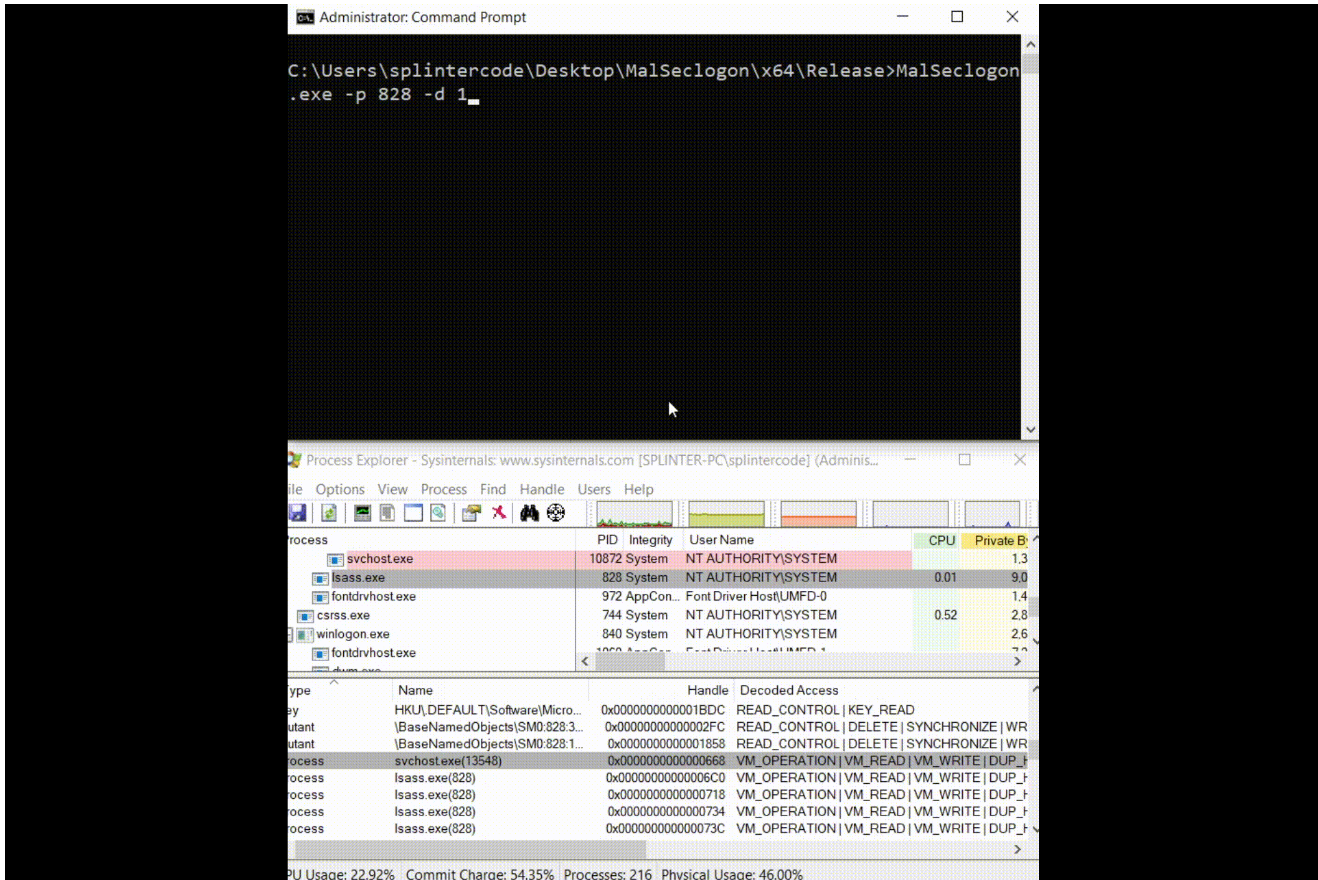
The magic happens of course in the **DuplicateHandle** function. The standard streams handles are duplicated from the hCaller process to the **newly** created process. We can control the hCaller with the **PID spoofing trick**. We can also control the **standard stream value** by using the @tiraniddo's discovery. So the two faults stack very nicely together and could **evict lsass handles** from lsass itself without interacting directly.

What's the plan to leak lsass process handles then?

1. Use **NtQuerySystemInformation** and get all the **process handle values** that resides in lsass, some of them are for lsass itself;
2. **Patch** the pid value in the current process TEB and specify the lsass PID;
3. Prepare the **CreateProcessWithLogonW** calls:
   - Specify the flag **LOGON_NETCREDENTIALS_ONLY** as the *dwLogonFlags* parameter;
   - Specify the flag **STARTF_USESTDHANDLES** in *lpStartupInfo->dwFlags*;
   - Specify the process handle values to **leak** from **lsass** in *lpStartupInfo->hStdInput*, *lpStartupInfo->hStdOutput* and *lpStartupInfo->hStdError*. So three at a time.
4. **Iterate** the CreateProcessWithLogonW calls until a **leaked lsass process handle** is found in the new process;
5. Enjoy the leaked lsass handle in the new process :D

Demo time!

Note: some breakpoints on the seclogon service has been added on windbg to demonstrate the whole process better



As shown in the above Demo, a **dump** of lsass is performed through a leaked handle and **saved** to the disk. In this case I have used the **MiniDumpWriteDump** function.

One thing we need to consider while using this function is that it tries to open a **new handle** to **lsass** and we definitely want to **avoid** that. The reason for this behavior seems to be inside **RtlQueryProcessDebugInformation** called internally by *MiniDumpWriteDump*, it opens a new handle to lsass instead of using the one **provided** in the call.
To solve this issue i have **hooked NtOpenProcess** before calling *MiniDumpWriteDump*. In the hooked function the **leaked lsass handle** is returned instead of **forwarding** the call to the kernel.
Another thing to take care of is to prevent the leaked handle from being closed by *RtlQueryProcessDebugInformation*. We could provide a **duplicate** of the leaked handle to solve the problem, but we want to avoid that a **registered kernel callback** is catching our dumping process for the newly duplicated handle.
One smarter thing to do is to use **SetHandleInformation** and protect the leaked handle from any **closing attempts** through the flag **HANDLE_FLAG_PROTECT_FROM_CLOSE**.

Cool! We finally managed to get a **new process** containing **leaked lsass handles** without invoking any OpenProcess or DuplicateHandle directly.

However the *MiniDumpWriteDump* call is still reading the lsass process memory **directly** and this is a very **noisy** operation.

One easy improvement for that would be using **unhooking** or **direct system calls** specifically for the **NtReadVirtualMemory** system call.
However, while they could be effective in most cases, they are not very effective against moderns EDR. These kind of techniques are flagged as malicious and the **EtwTi** monitoring would still catch the remote memory reads.

Any chance to do anything better?

Leveraging process address space cloning for indirect memory reads

Poking around in **ntoskrnl.exe** i have found a function that caught my attention for its name: **MiCloneProcessAddressSpace**. It has the following definition:

```
NTSTATUS MiCloneProcessAddressSpace(
        PEPROCESS ProcessToClone,
        PEPROCESS ProcessToInitialize)
```

What it does, briefly, is to create a **copy** of the specified "*ProcessToClone*" **address space** in the "*ProcessToInitialize*".
This is done by iterating through every **PTE** of the source process and **clone** them into the new process. All the **pages** in the new process are mapped as **shared copy-on-write**.

Cool! It seems it is what we need. It would be a nice idea to have a new "bridge" process that has the **same memory address space** of the real process and could allow us to **read** it **indirectly** without interacting with it.

But, how do we get there? Let's use a bottom-up approach.

The first cross reference returned by IDA is the function **MmInitializeProcessAddressSpace***:*

```
NTSTATUS MmInitializeProcessAddressSpace(
        PEPROCESS ProcessToInitialize,
        PEPROCESS ProcessToClone,
        PVOID SectionToMap,
        PULONG CreateFlags)
```

This function checks if the "**ProcessToClone**" parameter is provided and if that's the case it will invoke the function to clone the address space *MiCloneProcessAddressSpace*. We need to go to the upper caller...

There are 3 invocations of *MmInitializeProcessAddressSpace* from **PspAllocateProcess**. One of the invocations provides a "**ProcessToClone**" parameter **different from 0** to *MmInitializeProcessAddressSpace*. The value provided as the "ProcessToClone" in this specific invocation is the "**ParentObject**" for the **newly** created process. Below a snippet of reversed code from *PspAllocateProcess*:

```
else
{
  if ( !ParentObject )
    goto SkipProcessCloning;
  ProcessToInitialize->SectionBaseAddress = ParentObject->SectionBaseAddress;
  ReturnValue = MmInitializeProcessAddressSpace(ProcessToInitialize, ParentObject, 0i64, &CreateFlags);
```

The **else** branch code is **executed** if a **NULL SectionObject** is provided as a parameter to the function *PspAllocateProcess*. Then it **assigns** the parent process section base address to the new process section base address, so what we need. It starts to look interesting...

Going to the upper caller again we land into **PspCreateProcess**:

```
NTSTATUS PspCreateProcess(
      PHANDLE ProcessHandle,
      ACCESS_MASK DesiredAccess,
      POBJECT_ATTRIBUTES ObjectAttributes,
      HANDLE ParentProcess,
      KPROCESSOR_MODE PreviousMode,
      ULONG Flags,
      HANDLE SectionHandle,
      HANDLE DebugPort,
      HANDLE ExceptionPort)
```

One of the parameter provided to the function is the **SectionHandle** and this one is of our interest because it sets the **SectionObject** value:

If a **SectionHandle** is passed as a parameter to the function, the kernel tries to get the **SectionObject** from the **object manager**. If not it sets the SectionObject **to 0** (what triggers the chain for the process cloning).

Then it checks if the handle to the parent process provided holds the **PROCESS_CREATE_PROCESS** access and gets the parent process object (**ParentObject**) from the object manager:

```
if ( SectionHandle )
{
  SectionObject = 0i64;
  result = ObReferenceObjectByHandle(
             SectionHandle,
             SECTION_MAP_EXECUTE,
             MmSectionObjectType,
             PreviousMode,
             &SectionObject,
             0i64);
  if ( result < 0 )
    return result;
}
else
{
  SectionObject = 0i64;
  SectionObject = 0i64;
}
```

```
if ( !ParentProcess
  || (ntstatus = ObReferenceObjectByHandleWithTag(
                   ParentProcess,
                   PROCESS_CREATE_PROCESS,
                   (POBJECT_TYPE)PsProcessType,
                   PreviousMode,
                   'rCsP',
                   (PVOID *)&ParentObject,
                   0i64),
      ntstatus >= 0) )
```

We are almost there :)

Looking at the cross references of *PspCreateProcess* we can find the **NtCreateProcessEx** function:

```
NTSTATUS NtCreateProcessEx(
    PHANDLE ProcessHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    HANDLE ParentProcess,
    ULONG Flags,
    HANDLE SectionHandle,
    HANDLE DebugPort,
    HANDLE ExceptionPort,
    ULONG JobMemberLevel)
```

NtCreateProcessEx is a system call **exposed** by the kernel and can be invoked from a **userland process**. It turns out that the parameter "**SectionHandle**" from *NtCreateProcessEx* is passed directly to *PspCreateProcess*.

Cool!
All it requires to create a clone of a process is to **invoke** *NtCreateProcessEx* from *ntdll.dll* and provide the process handle (with *PROCESS_CREATE_PROCESS* access) into the **ParentProcess** parameter and **0** to the **SectionHandle** value.

NOTE: at the time of discovery i haven't found any public documentation about this particular usage of NtCreateProcessEx. Some time after, other researchers have published the same discovery independently as you can read here and here. And of course me being upset about the spoiler here :(
Duplicate happens...
Everything is set up to spawn a lsass process **clone** and read the memory from it.
One last thing to adjust is the **PROCESS_CREATE_PROCESS** access right **missing** from the **leaked lsass handle**. The duplicated handle by seclogon is missing one of the access to perform the *NtCreateProcessEx* call. Instead it contains the **duplicate** access (**PROCESS_DUP_HANDLE**) along with some other.

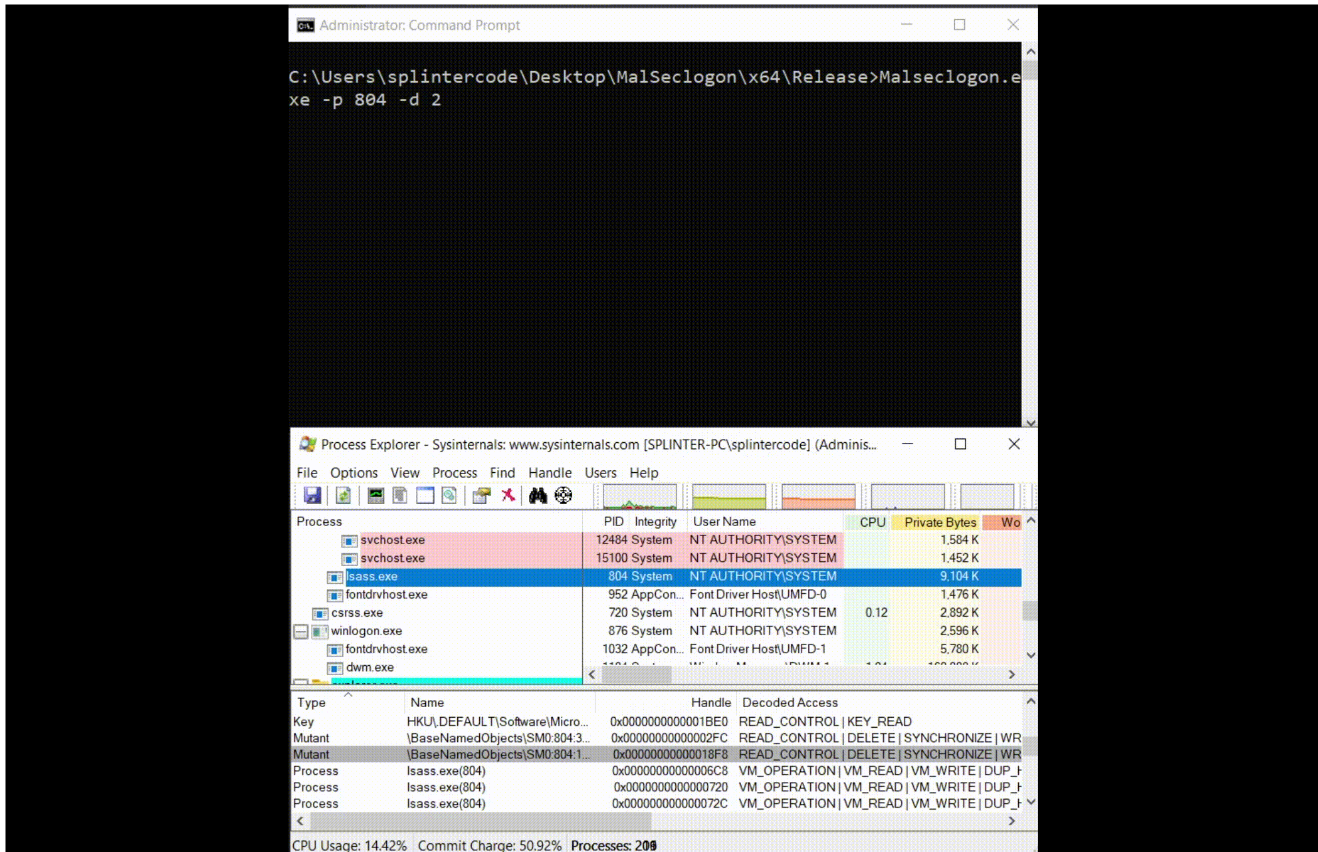
Using a trick by @tirannido described **here**, it's possible to get a **full access** process handle to lsass starting from the **leaked handle** as long as it holds the **duplicate** access. Explanation below:

"The DuplicateHandle system call has an interesting behaviour when using the pseudo current process handle, which has the value -1. Specifically if you try and duplicate the pseudo handle from another process you get back a full access handle to the source process." quote from **Bypassing SACL Auditing on LSASS**.

So what we need to do is to perform a *DuplicateHandle* call in this way:
DuplicateHandle((HANDLE)leakedHandle, (HANDLE)-1, GetCurrentProcess(), &hLeakedHandleFullAccess, 0, FALSE, DUPLICATE_SAME_ACCESS);

The newly duplicated process handle (**hLeakedHandleFullAccess**) will have enough access rights to perform the *NtCreateProcessEx* call in order to create the cloned process. We could potentially trigger some **registered kernel callback** on this specific *DuplicateHandle* call raised from our dumping process, but for now we are ok with that (maybe we could also avoid this? maybe a part 3 blogpost will come out :D).

Demo time!



POC

I have written **MalSeclogon**, a little tool to play with the seclogon service. It implements all the techniques shown in this blog post. You can find it **here**.

Conclusion

In this blog post it has been demonstrated how a series of **faults** and **bad assumptions** could make a windows system service **misbehaving** in a weird way. An attacker could take advantage of such behaviors to carry out **stealthier** operations.

In this specific scenario we have seen how an attacker can implement a very stealthy **memory dumping** technique by abusing an internal windows component: the **Secondary Logon Service**.
Defenders **shouldn't blindly trust** native windows applications, instead they should focus on the differences between normal and **abnormal** behavior... The seclogon service opening a new process handle to lsass is not something you see everyday ;)
This won't be the last blog post about the seclogon service. In the future posts of **"The hidden side of Seclogon"** series i will show other fancy stuff you can do by leveraging this service.

Stay tuned for more content about our beloved seclogon service :D

References