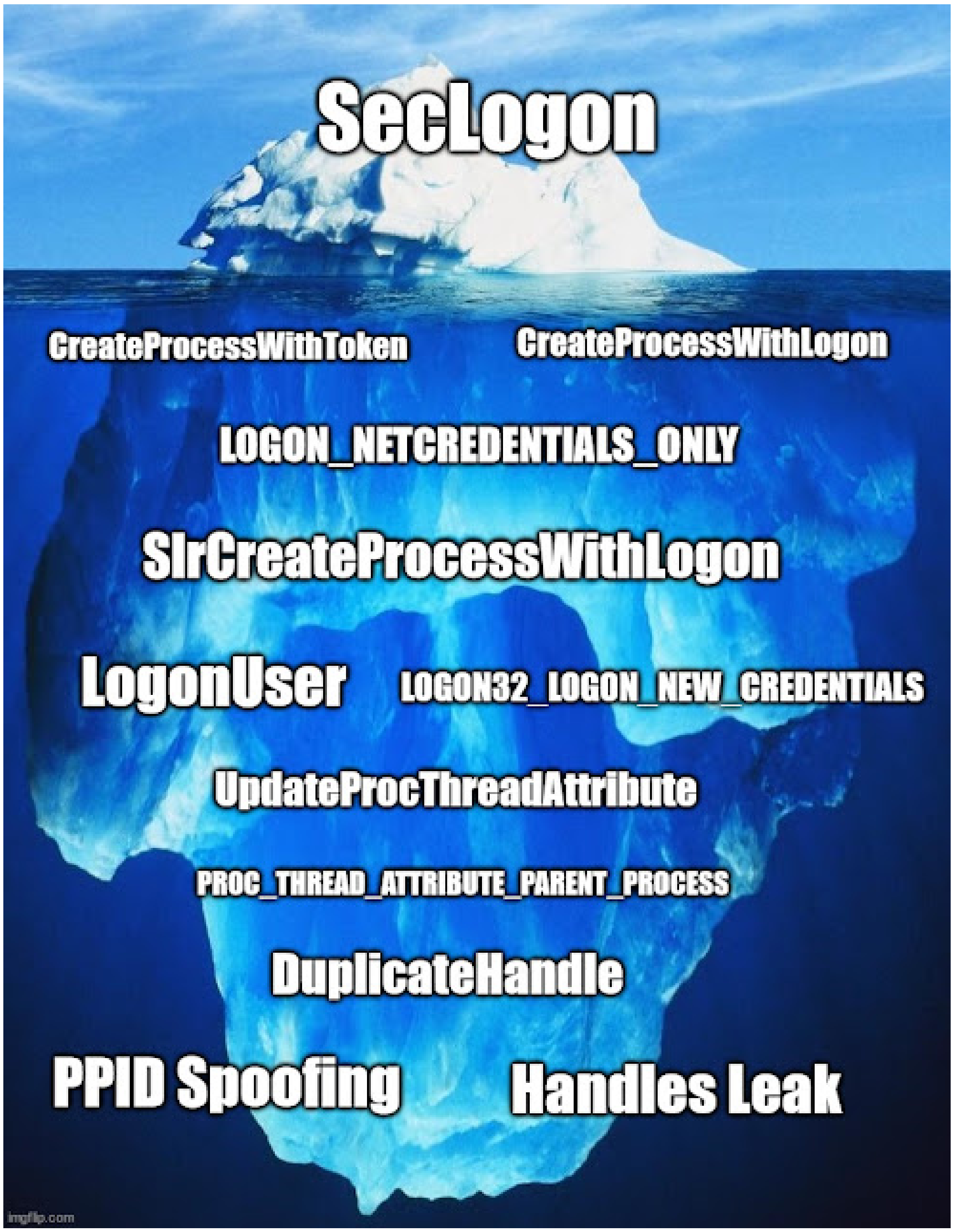
The hidden side of Seclogon part 3: Racing for LSASS dumps

splintercod3.blogspot.com/p/the-hidden-side-of-seclogon-part-3.html

by splinter_code - 28 June 2022



After my previous post "**The hidden side of Seclogon part 2: Abusing leaked handles to dump LSASS memory**" in which i described how to leverage the seclogon service in order to perform stealthier lsass dumps, i decided to continue this blog post series with another post about our beloved seclogon service, so here we are!

This blog post will cover, as promised, one of the mentioned point in the part 2:

"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)"

Just to recap, the seclogon service makes the bad assumption to determine the PID of the caller process by trusting the user input provided by the caller itself, i'm sure you know how this can go wrong.
A privileged attacker can exploit this behavior and can carry out stealthier operations like handle duplication and ppid spoofing.

In the previous post we observed how the seclogon implements all the operations required to expose the CreateProcessWithLogonW and CreateProcessWithTokenW functions, and more specifically implemented in the server function **SlrCreateProcessWithLogon**.

The first operation performed is an OpenProcess call to get a handle to the RPC caller by using a value under our control as the PID. The requested access is PROCESS_QUERY_INFORMATION | **PROCESS_CREATE_PROCESS** | PROCESS_DUP_HANDLE. This handle is opened also with the required access for the process cloning trick, more on that later.

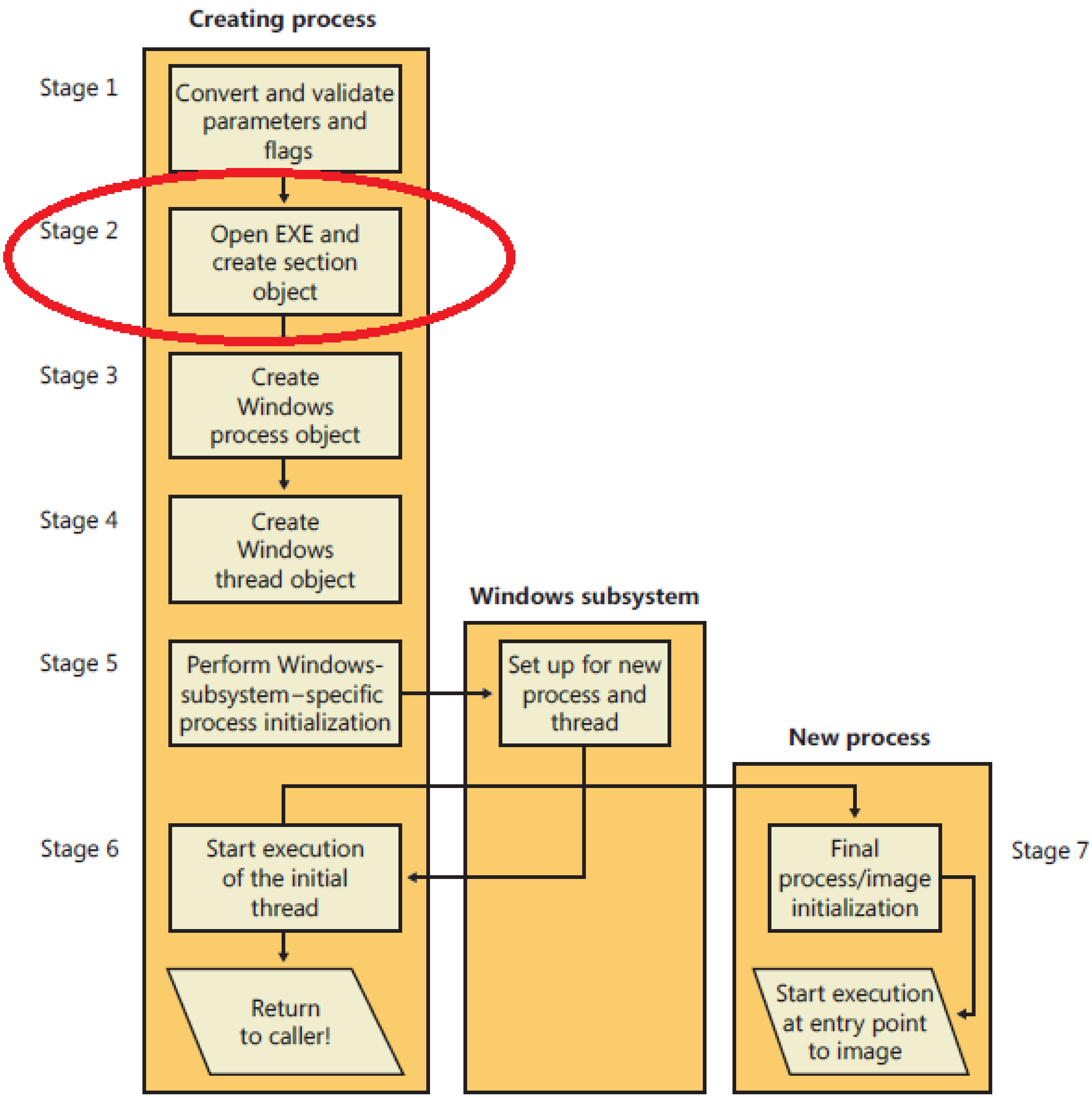
By spoofing our current process NtCurrentTeb()->ClientId->UniqueProcess value to the LSASS pid and then invoking the seclogon, we can trick the service into opening a handle to LSASS.
The problem with this handle is that it's closed shortly after its usage. Is there any way to delay this operation in order to give us enough time to duplicate this handle in our running process?

The best thing would be to find some operations involving files and set an OpLock on it to stop the execution flow and allow us to duplicate that handle before the CloseHandle call.
By inspecting all the code between the OpenProcess and CloseHandle calls, i couldn't find any file-related functions 😕

However, i noticed that one of the latest operations before the CloseHandle call was CreateProcessAsUser:

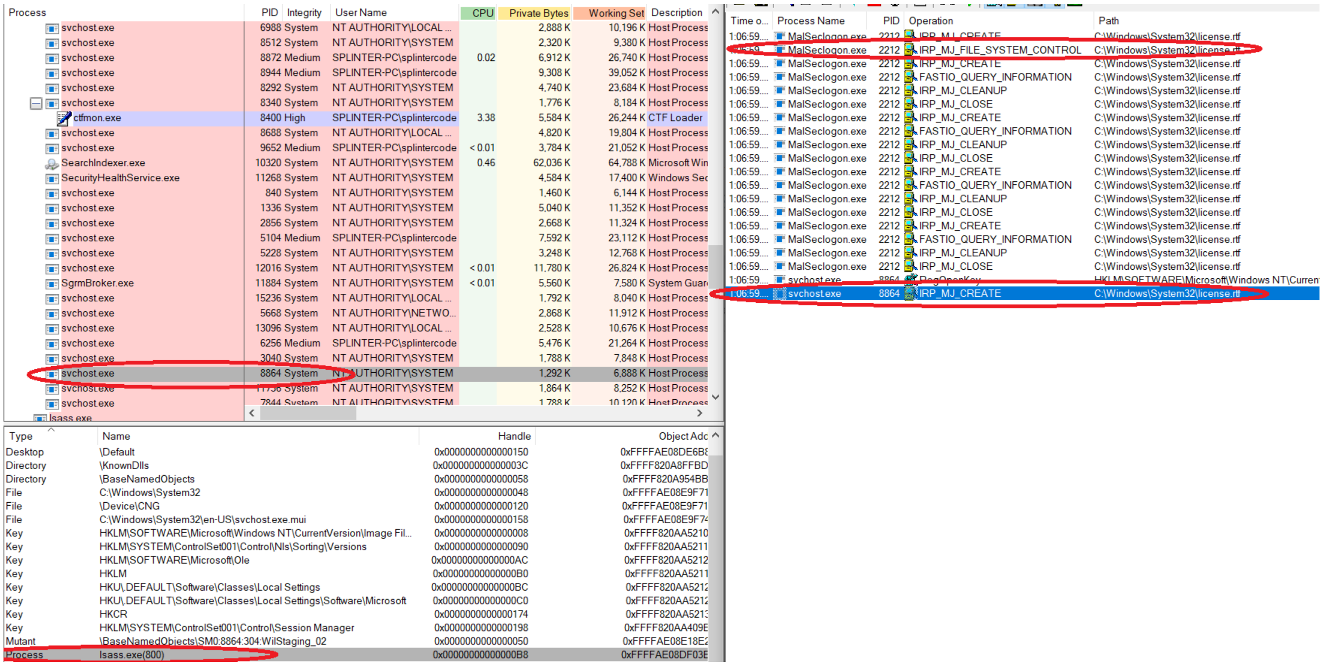
```
DWORD __fastcall SlrCreateProcessWithLogon(
        RPC_BINDING_HANDLE BindingHandle,
        PSECONDARYLOGONINFOW psli,
        LPPROCESS_INFORMATION ProcessInformationOutput)
{
 .
 .
 .
 hCaller = OpenProcess(
              PROCESS_QUERY_INFORMATION|PROCESS_CREATE_PROCESS|PROCESS_DUP_HANDLE,
              FALSE,
              psli->dwProcessId);
 if ( !hCaller )
   goto ReturnLastError;
 .
 .
 .
 if ( CreateProcessAsUserW(
        hToken,
        psli->lpApplicationName,
        psli->lpCommandLine,
        &defaultSecurityAttributes,
        &defaultSecurityAttributes,
        FALSE,
        dwCreationFlags | psli->dwCreationFlags,
        psli->lpEnvironment,
        psli->lpCurrentDirectory,
        psli->lpStartupInfo,
        ProcessInformationOutput) )
 .
 .
 .
 if ( hCaller )
   return CloseHandle(hCaller);
 return result;
```

CreateProcessAsUser allows to run a new process in the security context of the user represented by the specified token. Once some preparation steps are performed, it calls CreateProcessInternalW from kernel32.dll that will do all the dirty jobs of preparing the required data before going to the kernel (NtCreateUserProcess). One of the operation performed in the kernel is to open the provided file path and create the section object. Below a nice representation from the "Windows Internals part 1" book:



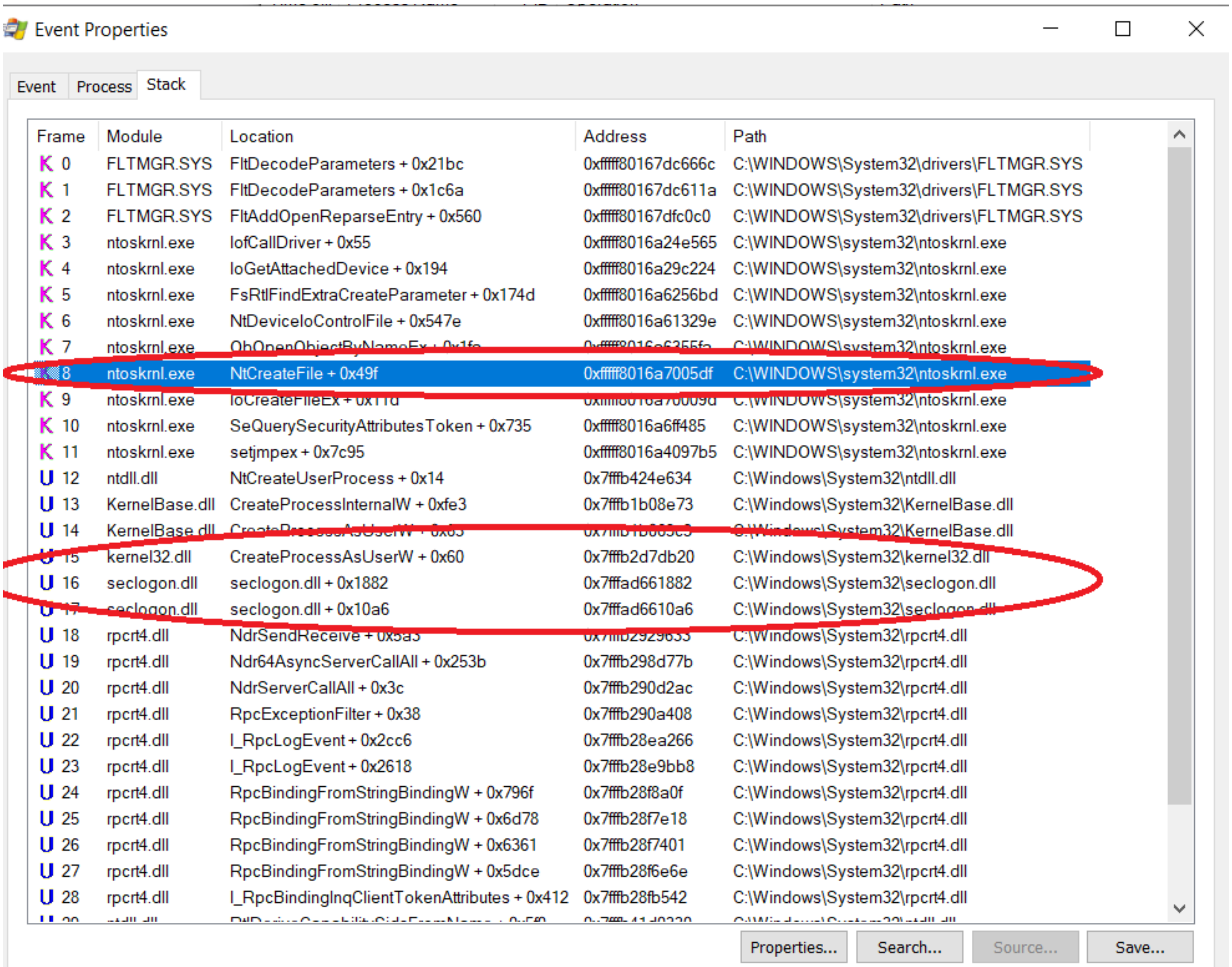
The main idea is to set an OpLock (thanks @tiraniddo) to a file under our control and then use that path as the input parameter for the create process function. In this way we expect that when the seclogon issues a CreateProcessAsUser call it will hit the oplock and will halt the process before it closes the lsass handle.
E.g. We can set an OpLock to "C:\Windows\System32\license.rtf" and then provide it as input to a CreateProcessWithLogonW call.

Seems a cool plan, let's try it out :D



As you can observe in the above screenshot, in the right capture of procmon the Malseclogon.exe process set an oplock to the "license.rtf" file and then shortly after we see the seclogon service (running under svchost.exe) trying to access to the file <-- here is when the lock condition happens. On the left side of procexp we can see that one process handle to lsass is still open in the seclogon process, ready to be duplicated :)

Great! Now we know we can lock the seclogon service for all the time required to duplicate the needed lsass handle. If you are wondering how the call stack looks like when the seclogon is locked, here you have it:



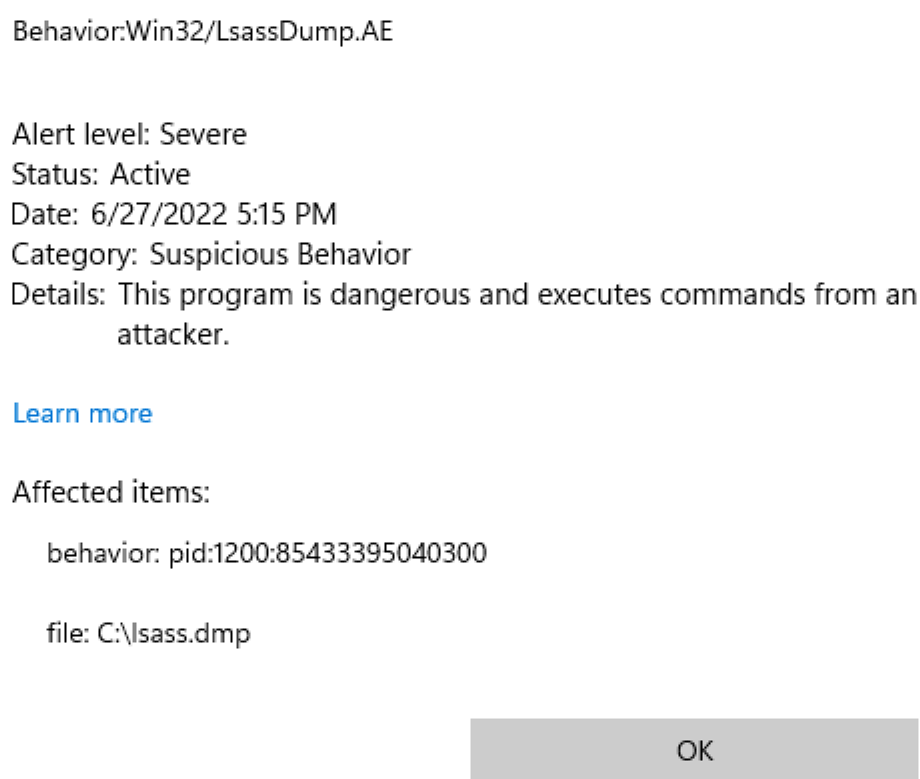
As a side note, even an unprivileged user can lock the seclogon service and the service won't be available for all users on the system ¯_(ツ)_/¯

Back to the point, we have everything needed to steal the leaked handle to lsass in this fun race:

1. Set an **OpLock** on "C:\Windows\System32\license.rtf";
2. Patch the pid value in the current process TEB and specify the **lsass PID**;
3. Use CreateProcessWithLogonW and specify "C:\Windows\System32\license.rtf" as the name of the module to be executed;
4. Wait the OpLock event to be signaled through GetOverlappedResults, this will occur once the seclogon tries to access our locked file;
5. Find the seclogon service pid. For bonus swag points i used a trick through NtQueryInformationFile described in my previous post, in this way i avoided to interact with the service control manager;
6. Enumerate all the process handles in the seclogon process through NtQuerySystemInformation;

7. Once a process handle to lsass is found, create an lsass clone through NtCreateProcessEx and use its handle in a call to MiniDumpWriteDump.
   Thanks to a trick by @__RastaMouse i avoided to hook NtOpenProcess like i did in the other dumping techniques. It turns out that by using 0 as the pid parameter of MiniDumpWriteDump it will do the job for preventing an additional open process to lsass;
8. Race won!

All good, all working, right? Yes, until i did the mistake to try this technique on a Windows 11 to check for newer compatibility:



On my Windows 11 vm i forgot to disable Windows Defender and of course it pestered me…

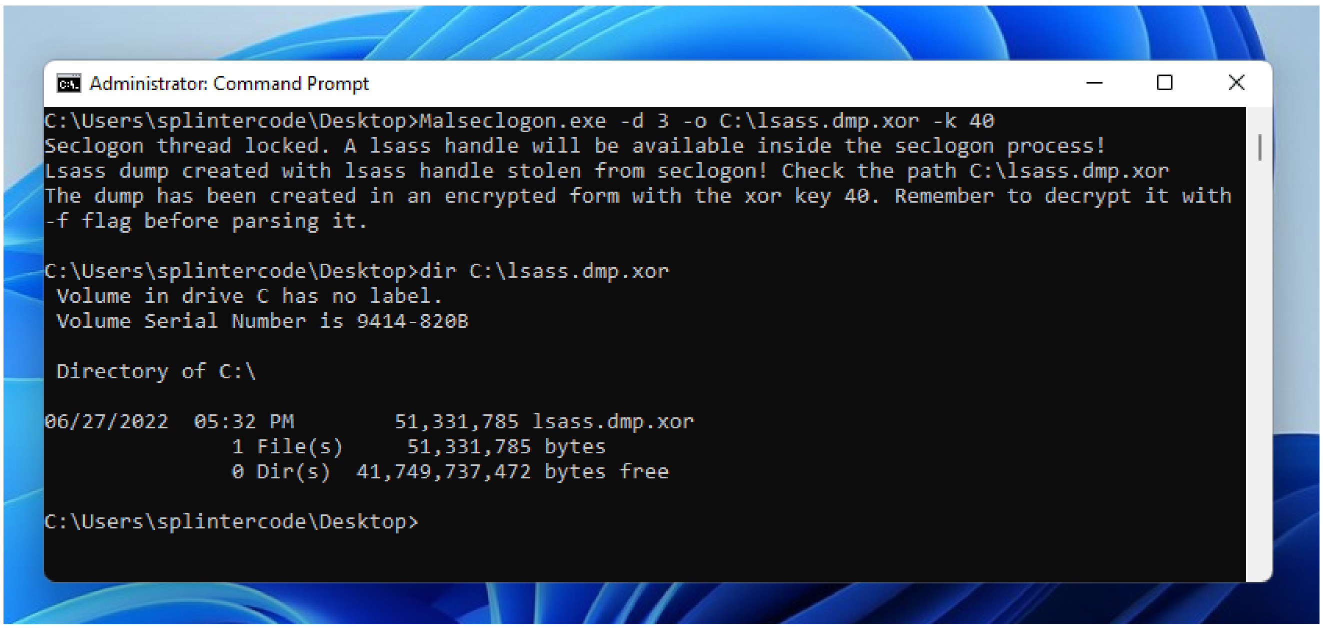
I usually dislike playing the cat and mouse game with these kinds of detection, but this way of detecting malicious stuff hurted my eyes too much, so i had to prove its uselessness.

Basically someone thought it was a good idea to detect lsass dumps by blocking the generated output file. Highly likely grepping some particular string + checking the MDMP header… What we can do is just XORing the dump content in memory before writing back to disk and then restore the content offline on another machine when we need to parse it and extract the credentials.

The first thing that came to my mind is to use a pipe and provide it in the MiniDumpWriteDump function as the file handle parameter. However it seemed a bit dirty and MiniDumpWriteDump allows a cleaner way to do it through a minidump callback.

Luckily, i found a working and ready to copy-paste code here that does the job.
Once the lsass memory content is dumped into our memory process we simply apply a 1-byte Xor encryption to the content before writing back to the disk.

Putting it all together, finally i got the dumping technique through leaked handle and race condition fully working:



I have released this new dumping technique in the Malseclogon repo -->
https://github.com/antonioCoco/MalSeclogon

That's all folks :)
This is the last post about lsass dumping related to the seclogon service.
In the next post of "The hidden side of Seclogon" series i will cover the part 1 that will mainly explain all of the cool code behind RunasCs :D

If you are curious to read about the enhancement i did to the PPID spoofing feature in Malseclogon, feel free to read the bonus section below 👇 of course a bit unrelated to LSASS dumping.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

[BONUS SECTION] - PPID spoofing running as SYSTEM

In the previous post we have seen how to leverage the seclogon service in order to perform a PPID spoofing primitive. This was achieved by using CreateProcessWithLogonW with the magic flag LOGON_NETCREDENTIALS_ONLY. This will allow to run a child process running in the security context of the rpc caller.

One desired outcome when performing PPID spoofing is to spawn a process with the same privileges of the parent and that wasn't the case while abusing CreateProcessWithLogonW.

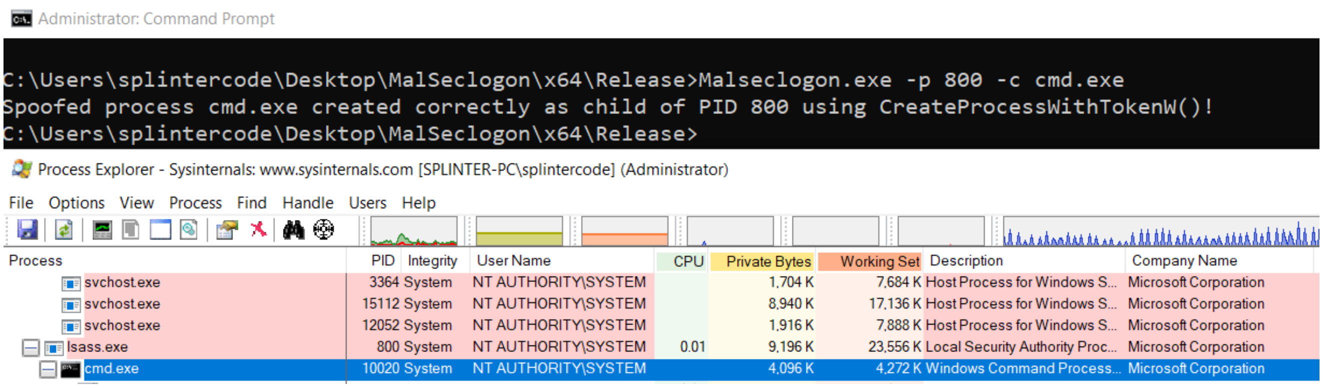
However, the CreateProcessWithTokenW function allows to specify a token as input parameter. Could we leverage this other function in the seclogon? Let's reverse how this is implemented:

```
if ( RpcImpersonateClient(BindingHandleTmp) )
  goto ReturnLastError;
flagIsImpersonating = 1;
flagIsImpersonating2 = 1;
if ( psli->hToken )                          // this is true when using CreateProcessWithTokenW
{
  if ( !OpenProcessToken(hCaller, 0xCu, &ClientToken) )
    goto ReturnLastError;
  RequiredPrivileges.PrivilegeCount = 1;
  RequiredPrivileges.Privilege[0].Luid = SE_IMPERSONATE_PRIVILEGE;
  if ( !PrivilegeCheck(
          ClientToken,
          &RequiredPrivileges,
          &pfResult) )
    pfResult = 0;
  CloseHandle(ClientToken);
  if ( !pfResult )
  {
    LastError = ERROR_PRIVILEGE_NOT_HELD;
    goto SetFlagAndClearVariablesForExit;
  }
  CurrentProcess = GetCurrentProcess();
  if ( !DuplicateHandle(
          hCaller,
          psli->hToken,
          CurrentProcess,
          &hToken,
          0,
          0,
          DUPLICATE_SAME_ACCESS) )
    goto ReturnLastError;
  LastError = RpcRevertToSelfEx(BindingHandleTmp);
```

Basically, the seclogon firstly impersonates the rpc caller. Then it checks if it holds the Impersonation privilege. If that's the case it duplicates the token handle from the rpc caller to the seclogon service. Considering that the rpc caller is under our control with the spoofing trick, we could use a token inside the parent we want to spoof, of course if it exists.
Then, the duplicated token is used in a CreateProcessAsUserW call to spawn the child process:

```
if ( CreateProcessAsUserW(
        hToken,
        psli->lpApplicationName,
        psli->lpCommandLine,
        &defaultSecurityAttributes,
        &defaultSecurityAttributes,
        FALSE,
        dwCreationFlags | psli->dwCreationFlags,
        psli->lpEnvironment,
        psli->lpCurrentDirectory,
        psli->lpStartupInfo,
        ProcessInformationOutput) )
```

The idea here is to try to specify a token handle residing in the process we want to spoof and see if we inherits either the primary token of the process and the spoofed parent itself:



And done, the child process is running with the token of the parent :D