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

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

by splinter_code - 28 June 2022



After my previous post "<u>The hidden side of Seclogon part 2: Abusing leaked handles</u> <u>to dump LSASS memory</u>" 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 <u>CreateProcessWithLogonW</u> and <u>CreateProcessWithTokenW</u> functions, and more specifically implemented in the server function **SlrCreateProcessWithLogon**.

The first operation performed is an <u>OpenProcess</u> 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 <u>CreateProcessAsUser</u>:

```
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 ( !<mark>hCaller</mark> )
   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 <u>OpLock</u> (thanks <u>@tiraniddo</u>) 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

Process		PID Integrity	User Name	CPU	Private Bytes	Working Set Description	r	• • •		
svchost	texe	6988 System	NT AUTHORITY/LOCAL		2.888 K	10,196 K Host Process	Time o	Process Name	PID Operation	Path
svchost	texe	8512 System	NT AUTHORITY\SYSTEM		2.320 K	9.380 K Host Process	1:06:59	MalSeclogon exe		C-Windows\System32\license.rtf
svchost	texe	8872 Medium	SPLINTER-PC\splintercode	0.02	6.912 K	26,740 K Host Process	COLCEQ.	MalSeclogon.exe	2212 KIRP_MJ_FILE_SYSTEM_CONTROL	C:\Windows\System32\license #
svchost	texe	8944 Medium	SPLINTER-PC\splintercode		9.308 K	39.052 K Host Process	1:06:59	MalSeclogon.exe	2212 KIRP_MJ_CREATE	C:\Windows\System32\license.rtt
svchost	texe	8292 System	NT AUTHORITY/SYSTEM		4,740 K	23.684 K Host Process	1.00.59	MalSeclogon.exe	2212 KEASTIO_QUERT_INFORMATION	C://windows/System32/license.tf
svchost	texe	8340 System	NT AUTHORITY\SYSTEM		1.776 K	8.184 K Host Process	1:06:59	MalSeclogon.exe	2212 REP MJ CLOSE	C:\Windows\System32\license.tt
ctfmo	on.exe	8400 High	SPLINTER-PC\splintercode	3.38	5.584 K	26.244 K CTF Loader	1:06:59	ThalSeclogon.exe	2212 IRP MJ CREATE	C:\Windows\System32\license.rtf
svchost	texe	8688 System	NT AUTHORITY/LOCAL		4.820 K	19.804 K Host Process	1:06:59	MalSeclogon.exe	2212 KEASTIO_QUERY_INFORMATION	C:\Windows\System32\license.rtf
svchost	texe	9652 Medium	SPLINTER-PC\splintercode	< 0.01	3.784 K	21.052 K Host Process	1:06:59	MalSeclogon.exe	2212 KIRP_MJ_CLEANUP	C:\Windows\System32\license.rtf
.Q. Searchl	Indexer exe	10320 System	NT AUTHORITY/SYSTEM	0.46	62.036 K	64.788 K Microsoft Win	1:06:59	📧 MalSeclogon.exe	2212 SIRP_MJ_CLOSE	C:\Windows\System32\license.rtf
Security	vHealthService exe	11268 System	NT AUTHORITY/SYSTEM		4.584 K	17400 K Windows Sec	1:06:59	MalSeclogon.exe	2212 KIRP_MJ_CREATE	C:\Windows\System32\license.rtf
svchost	tere	840 System	NT AUTHORITY/SYSTEM		1.460 K	6 144 K Host Process	1:06:59	MalSeclogon.exe	2212 KEASTIO_QUERY_INFORMATION	C:\Windows\System32\license.rtf
svchost	tere	1336 System	NT AUTHORITY/SYSTEM		5 040 K	11 352 K Host Process	1:06:59	MalSeclogon.exe	2212 KIRP_MJ_CLEANUP	C:\Windows\System32\license.rtf
svchost	tere	2856 System	NT AUTHORITY/SYSTEM		2 668 K	11 324 K Host Process	1:06:59	MalSeclogon.exe	2212 KIRP_MJ_CLOSE	C:\Windows\System32\license.rtf
sychost	tere	5104 Medium	SPLINTER-PC\splintercode		7 592 K	23 112 K Host Process	1.06.59	MalSeclogon.exe		C://windows/System32/license.nt
evchoet	tore	5228 Svetom	NT ALITHODITY/SYSTEM		3 248 K	12 768 K Host Process	1.06.59	MalSeclogon.exe		C:\Windows\System32\license.tt
evchoet	toxe	12016 System		< 0.01	11 780 K	26.824 K Host Process	1:06:59	MalSeclogon.exe	2212 REP MJ CLOSE	C:\Windows\System32\license.tf
SomBr	roker exe	11884 System		< 0.01	5 560 K	7.580 K System Guan	1:06:59	sychostexe	8864 R DogOposKov	HKI MISOFTWARE/Microsoft/Windows NT/Curren
evchoet	toxo	15236 System		< 0.01	1 792 K	8 040 K Host Process	1:06:59	svchostexe	8864 WIRP_MJ_CREATE	C:\Windows\System32\license.tt
SVCHOSE avehant	texe	ECC System			1,752 K	11 012 K Heat Dresses				
SVCHOSE avehant	texe	12006 System			2,000 K	10.676 K Heat Dresses				
svchost	d eve	60EC Medium	COUNTED DOlaristoreade		2,520 K	21 264 K Heat Dresses				
svchost	toro	3040 System	NT ALITHODITY/SYSTEM		1 788 K	7 8/8 K Host Process				
svehost	tore	8864 System	NT AUTHORIT (STSTEM		1,700 K	6 888 K Host Process				
SVCHOSE	stexe	ooo4 System	NT AUTHORIT (STSTEM		1,252 K	0,000 K Host Process				
svchost.	d ave	7844 System			1,004 K	10 120 K Heat Preserve				
Svenost.	rexe	<	NT ALTEORIT DATATEM		1 /00 8	>				
Telepides Martine Control Cont										
Dealstern	Default		0000	F	00150					
Directory	sktop (Default UxU				00150	0xFFFFAE00DE0DC				
Directory	irectory (KnownDils 0x				00050	0xFFFF820A954BB				
File	e C:\Windows\System32				00048	0xFFFFAE08F9E71				
File	e \Device\CNG 0x				00120	0xFFFFAE08E9F71				
File	ile C:\Windows\Svstem32\en-US\svchost.exe.mui 0x0				00158	0xFFFFAE08E9F74				
Key HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image Fil 0x			Version\Image Fil 0x000	x000000000000008		0xFFFF820AA5210				
Key	y HKLM\SYSTEM\ControlSet001\Control\NIs\Sorting\Versions 0			0x000000000000090		0xFFFF820AA5211				
Key	HKLM\SOFTWARE\Microsoft\Ole 0x0			0000000000000AC		0xFFFF820AA5212				
Key HKLM Ox			0x000	0x000000000000000000000000000000000000		0xFFFF820AA5211				
Key HKULDEFAULTISoftware(Classes)Local Settings 0x			0x000	x0000000000000BC		0xFFFF820AA5212				
Key HKCD HKCD http://www.commare.classes/Local Settings/Software/Microsoft Ux			onware\wiicrosoft 0x000	000000000000000000000000000000000000000		0xFFFF820AA5212				
Key HKI MISYSTEM/ControlSet/01\ControlSession Manager 0			00000 0x000	000000000000198		0xFFFF82044409F				
Mutant \BaseNamedObiects\SM0/8864/304/WilStaging 02			0x000	000000000000000000000000000000000000000		0xFFFFAE08E18E2				
Process	Isass.exe(800)		0x000	000000000	000B8	0xFFFFAE08DF03E				
			011000							

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:

💐 Event Properties

Event Process Stack

Frame	Module	Location	Address	Path	
K 0	FLTMGR.SYS	FltDecodeParameters + 0x21bc	0xffff80167dc666c	C:\WINDOWS\System32\drivers\FLTMGR.SYS	
K 1	FLTMGR.SYS	FltDecodeParameters + 0x1c6a	0xffff80167dc611a	C:\WINDOWS\System32\drivers\FLTMGR.SYS	
K 2	FLTMGR.SYS	FltAddOpenReparseEntry + 0x560	0xffff80167dfc0c0	C:\WINDOWS\System32\drivers\FLTMGR.SYS	
K 3	ntoskrnl.exe	lofCallDriver + 0x55	0xffff8016a24e565	C:\WINDOWS\system32\ntoskrnl.exe	
K 4	ntoskrnl.exe	loGetAttachedDevice + 0x194	0xffff8016a29c224	C:\WINDOWS\system32\ntoskrnl.exe	
K 5	ntoskrnl.exe	FsRtlFindExtraCreateParameter + 0x174d	0xffff8016a6256bd	C:\WINDOWS\system32\ntoskrnl.exe	
K 6	ntoskrnl.exe	NtDeviceIoControlFile + 0x547e	0xffff8016a61329e	C:\WINDOWS\system32\ntoskrnl.exe	
Κ7	ntoskrnl.exe	ObOnenObjectByNamoEx + 0x1fe	0vffff9016-6255fo	C:\WINDOWS\system32\ntoskrnl.exe	
8	ntoskrnl.exe	NtCreateFile + 0x49f	0xffff8016a7005df	C:\WINDOWS\system32\ntoskrnl.exe	
K 9	ntoskrnl.exe	IOCreateFileEx + UX I TO	000000000000000000000000000000000000000	C.\windows\system32\ntoskrnl.exe	
K 10	ntoskrnl.exe	SeQuerySecurityAttributesToken + 0x735	0xffff8016a6ff485	C:\WINDOWS\system32\ntoskrnl.exe	
K 11	ntoskrnl.exe	setjmpex + 0x7c95	0xffff8016a4097b5	C:\WINDOWS\system32\ntoskrnl.exe	
U 12	ntdll.dll	NtCreateUserProcess + 0x14	0x7fffb424e634	C:\Windows\System32\ntdll.dll	
U 13	KernelBase.dll	CreateProcessInternalW + 0xfe3	0x7fffb1b08e73	C:\Windows\System32\KernelBase.dll	
U 14	KernelBase dll	Crosto Deservation - 0x00	UX7111D TD000C0	0.(Windows)System32\KernelBase.dll	
5 15	kernel32.dll	CreateProcessAsUserW + 0x60	0x7fffb2d7db20	C:\Windows\System32\kernel32.dll	
U 16	seclogon.dll	seclogon.dll + 0x1882	0x7fffad661882	C:\Windows\System32\seclogon.dll	
0 17	<u>soclogon.dll</u>	seclogon.dll + 0x10a6	0x7fffad6610a6	C:\Windows\System32\seclogen.dll	
U 18	rpcrt4.dll	NdrSendReceive + 0x5a5	07/1102929033	C:\Windows\System32\rpcrt4.dll	
U 19	rpcrt4.dll	Ndr64AsyncServerCallAll + 0x253b	0x7fffb298d77b	C:\Windows\System32\rpcrt4.dll	
U 20	rpcrt4.dll	NdrServerCallAll + 0x3c	0x7fffb290d2ac	C:\Windows\System32\rpcrt4.dll	
U 21	rpcrt4.dll	RpcExceptionFilter + 0x38	0x7fffb290a408	C:\Windows\System32\rpcrt4.dll	
U 22	rpcrt4.dll	I_RpcLogEvent + 0x2cc6	0x7fffb28ea266	C:\Windows\System32\rpcrt4.dll	
U 23	rpcrt4.dll	I_RpcLogEvent + 0x2618	0x7fffb28e9bb8	C:\Windows\System32\rpcrt4.dll	
U 24	rpcrt4.dll	RpcBindingFromStringBindingW + 0x796f	0x7fffb28f8a0f	C:\Windows\System32\rpcrt4.dll	
U 25	rpcrt4.dll	RpcBindingFromStringBindingW + 0x6d78	0x7fffb28f7e18	C:\Windows\System32\rpcrt4.dll	
U 26	rpcrt4.dll	RpcBindingFromStringBindingW + 0x6361	0x7fffb28f7401	C:\Windows\System32\rpcrt4.dll	
U 27	rpcrt4.dll	RpcBindingFromStringBindingW + 0x5dce	0x7fffb28f6e6e	C:\Windows\System32\rpcrt4.dll	
U 28	rpcrt4.dll	$I_RpcBindingInqClientTokenAttributes + 0x412$	0x7fffb28fb542	C:\Windows\System32\rpcrt4.dll	
		DilDesite One - Hilbs Of Ja Farme Marrier & 0.00	0.700-41-00000	0.1147 1 10 1 001 1 11 11	

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 $(\gamma)/$

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 <u>CreateProcessWithLogonW</u> 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 <u>GetOverlappedResults</u>, 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 <u>NtQueryInformationFile</u> described in my <u>previous post</u>, in this way i avoided to interact with the service control manager;
- 6. Enumerate all the process handles in the seclogon process through <u>NtQuerySystemInformation;</u>

 \times

- 7. Once a process handle to lsass is found, create an lsass clone through NtCreateProcessEx and use its handle in a call to <u>MiniDumpWriteDump</u>. Thanks to a <u>trick</u> by <u>@_RastaMouse</u> i avoided to hook NtOpenProcess like i did in the other dumping techniques. It turns out that by using o 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:

Behavior:Win32/LsassDump.AE Alert level: Severe Status: Active Date: 6/27/2022 5:15 PM Category: Suspicious Behavior Details: This program is dangerous and executes commands from an attacker.

Affected items:

behavior: pid:1200:85433395040300

file: C:\lsass.dmp

OK

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 <u>here</u> 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 --> <u>https://github.com/antonioCoco/MalSeclogon</u>

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 <u>RunasCs</u> :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 \uparrow 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:

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:

Administrator: Command Prompt										
C:\Users\splintercode\Desktop\MalSeclogon\x64\Release>Malseclogon.exe -p 800 -c cmd.exe Spoofed process cmd.exe created correctly as child of PID 800 using CreateProcessWithTokenW()! C:\Users\splintercode\Desktop\MalSeclogon\x64\Release>										
2 Process Explorer - Sysinternals: www.sysinternals.com [SPLINTER-PC\splintercode] (Administrator)										
File Options View Process Find Handle Users Help										
🛃 🛃 📰 🗈 🗖 🚳 🚰 🗡 🏘 😨	-Au				A dista dan data da a	1				
Process	PID Integrity	User Name	CPU	Private Bytes	Working Set Description	Company Name				
svchost.exe	3364 System	NT AUTHORITY\SYSTEM		1,704 K	7,684 K Host Process for Windows S.	Microsoft Corporation				
svchost.exe	15112 System	NT AUTHORITY\SYSTEM		8,940 K	17,136 K Host Process for Windows S.	Microsoft Corporation				
svchost.exe	12052 System	NT AUTHORITY\SYSTEM		1,916 K	7,888 K Host Process for Windows S.	Microsoft Corporation				
🔄 🔲 Isass.exe	800 System	NT AUTHORITY\SYSTEM	0.01	9,196 K	23,556 K Local Security Authority Proc.	Microsoft Corporation				
- Cmd.exe	10020 System	NT AUTHORITY\SYSTEM		4,096 K	4,272 K Windows Command Process	Microsoft Corporation				

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