## Hiding In PlainSight - Indirect Syscall is Dead! Long Live Custom Call Stacks

Oxdarkvortex.dev/hiding-in-plainsight/

Posted on 29 Jan 2023 by Paranoid Ninja

## NOTE: This is a PART II blog on Stack Tracing evasion. PART I can be found here.

This is the second part of the blog I wrote 3 days back on proxying DLL loads to hide suspicious stack traces leading to a user allocated RX region. I won't be going in depth on how stack works, because I already covered that in the previous blog which can be accessed from the above link. We previously saw that we can manipulate the call and jmp instructions to request windows callbacks into calling LoadLibrary API call. However, stack tracing detections go far beyond just hunting DLL loads. When you inject a reflective DLL into local or remote process, you have to call API calls such as VirtualAllocEx/VirtualProtectEx which indirectly calls NtAllocateVirtualMemory/NtProtectVirtualMemory. However, when you check the call stack of the legitimate API calls, you will notice that WINAPIs like VirtualAlloc/VirtualProtect are mostly called by non-windows DLL functions. Majority of windows DLLs will call NtAllocateVirtualMemory/NtProtectVirtualMemory directly. Below is a quick example of the callstack for NtProtectVirtualMemory when you call RtlallocateHeap.

## 🔳 Stack - thread 2528

	Name
0	ntdll.dll!NtAllocateVirtualMemory
1	ntdll.dll!RtlProtectHeap+0x635
2	ntdll.dll!RtlProtectHeap+0x29b
3	ntdll.dll!RtlAllocateHeap+0x325a
4	ntdll.dll!RtlAllocateHeap+0xaad

This means that since ntdll.dll is not dependent on any other DLL, all functions in ntdll which require playing around with permissions for memory regions will call the NTAPIs directly. Thus, it means that if we are able to reroute our NtAllocateVirtualMemory call via a clean stack from ntdll.dll itself, we wont have to worry about detections at all. Most red teams rely on indirect syscalls to avoid detections. In case of indirect syscalls, you simply jump to the address of syscall instruction after carefully creating the stack, but the issue here is that indirect syscalls will only change the return address for the syscall instruction in ntdll.dll.

Return Address in this case is the location where the syscall instruction needs to return to, after the syscall is complete. But the rest of the stack below the return address will still be suspicious as they emerge out from the RX region. If an EDR checks the full stack of the NTAPI, it can easily identify that the return address eventually reaches back to the user allocated RX region. This means, a return address to ntdll.dll region, but stack originating from RX region is a 100% anomaly with zero chances of being a false positive. This is an easy win for EDRs who utilize ETW for syscall tracing in the kernel.

Thus in order to evade this, I spent some time reversing several ntdll.dll functions and found that with a little bit of assembly knowledge and how windows callbacks work, we should be able to manipulate the callback into calling any NTAPI function. For this blog, we will take an example of NtAllocateVirtualMemory and we will pick the code from our part I blog and modify it. We will take an example of the same API TpAllocWork which can execute a call back function. But instead of passing on a pointer to a string like we did in the case of Dll Proxying, we will pass on a pointer to a structure this time. We will also avoid any global variables this time by making sure all the necessary information goes within the struct as we cannot have global variables when we write our shellcodes. The definition of NtAllocateVirtualMemory as per msdn is:

```
___kernel_entry NTSYSCALLAPI NTSTATUS NtAllocateVirtualMemory(
 [in]
           HANDLE
                      ProcessHandle,
 [in, out] PVOID
                      *BaseAddress,
 [in]
           ULONG_PTR ZeroBits,
 [in, out] PSIZE_T
                      RegionSize,
                      AllocationType,
 [in]
           ULONG
 [in]
           ULONG
                     Protect
);
```

This means, we need to pass on a pointer for NtAllocateVirtualMemory and its arguments inside a structure to the callback so that our callback can extract these information from the structure and execute it. We will ignore the arguments which stay static such as ULONG\_PTR ZeroBits which is always zero and ULONG AllocationType which is always MEM\_RESERVE | MEM\_COMMIT which in hex is 0x3000. Thus adding in the remaining arguments, the structure will look like this:

```
typedef struct _NTALLOCATEVIRTUALMEMORY_ARGS {
    UINT_PTR pNtAllocateVirtualMemory; // pointer to NtAllocateVirtualMemory - rax
    HANDLE hProcess; // HANDLE ProcessHandle - rcx
    PVOID* address; // PVOID *BaseAddress - rdx; ULONG_PTR
ZeroBits - 0 - r8
    PSIZE_T size; // PSIZE_T RegionSize - r9; ULONG
AllocationType - MEM_RESERVE|MEM_COMMIT = 3000 - stack pointer
    ULONG permissions; // ULONG Protect - PAGE_EXECUTE_READ - 0x20
- stack pointer
} NTALLOCATEVIRTUALMEMORY_ARGS, *PNTALLOCATEVIRTUALMEMORY_ARGS;
```

We will then initialize the structure with the required arguments and pass it as a pointer to TpAllocWork and call our function WorkCallback which is written in assembly.

```
#include <windows.h>
#include <stdio.h>
typedef NTSTATUS (NTAPI* TPALLOCWORK)(PTP_WORK* ptpWrk, PTP_WORK_CALLBACK
pfnwkCallback, PVOID OptionalArg, PTP_CALLBACK_ENVIRON CallbackEnvironment);
typedef VOID (NTAPI* TPPOSTWORK)(PTP_WORK);
typedef VOID (NTAPI* TPRELEASEWORK)(PTP_WORK);
typedef struct _NTALLOCATEVIRTUALMEMORY_ARGS {
    UINT_PTR pNtAllocateVirtualMemory; // pointer to NtAllocateVirtualMemory - rax
    HANDLE hProcess;
                                        // HANDLE ProcessHandle - rcx
    PVOID* address;
                                       // PVOID *BaseAddress - rdx; ULONG_PTR
ZeroBits - 0 - r8
                                         // PSIZE_T RegionSize - r9; ULONG
    PSIZE_T size;
AllocationType - MEM_RESERVE | MEM_COMMIT = 3000 - stack pointer
    ULONG permissions;
                                        // ULONG Protect - PAGE_EXECUTE_READ - 0x20
- stack pointer
} NTALLOCATEVIRTUALMEMORY_ARGS, *PNTALLOCATEVIRTUALMEMORY_ARGS;
extern VOID CALLBACK WorkCallback(PTP_CALLBACK_INSTANCE Instance, PVOID Context,
PTP_WORK Work);
int main() {
    LPVOID allocatedAddress = NULL;
    SIZE_T allocatedsize = 0x1000;
    NTALLOCATEVIRTUALMEMORY_ARGS ntAllocateVirtualMemoryArgs = { 0 };
    ntAllocateVirtualMemoryArgs.pNtAllocateVirtualMemory = (UINT_PTR)
GetProcAddress(GetModuleHandleA("ntdll"), "NtAllocateVirtualMemory");
    ntAllocateVirtualMemoryArgs.hProcess = (HANDLE)-1;
    ntAllocateVirtualMemoryArgs.address = &allocatedAddress;
    ntAllocateVirtualMemoryArgs.size = &allocatedsize;
    ntAllocateVirtualMemoryArgs.permissions = PAGE_EXECUTE_READ;
    FARPROC pTpAllocWork = GetProcAddress(GetModuleHandleA("ntdll"), "TpAllocWork");
    FARPROC pTpPostWork = GetProcAddress(GetModuleHandleA("ntdll"), "TpPostWork");
    FARPROC pTpReleaseWork = GetProcAddress(GetModuleHandleA("ntdll"),
"TpReleaseWork");
    PTP_WORK WorkReturn = NULL;
    ((TPALLOCWORK)pTpAllocWork)(&WorkReturn, (PTP_WORK_CALLBACK)WorkCallback,
&ntAllocateVirtualMemoryArgs, NULL);
    ((TPPOSTWORK)pTpPostWork)(WorkReturn);
    ((TPRELEASEWORK)pTpReleaseWork)(WorkReturn);
    WaitForSingleObject((HANDLE)-1, 0x1000);
    printf("allocatedAddress: %p\n", allocatedAddress);
    getchar();
   return 0;
}
```

Now this is where things get interesting. In case of DLL proxy, we executed LoadLibrary with only one argument i.e. the name of the DLL to load which is passed on to the RCX register. But in the case of NtAllocateVirtualMemory, we have a total of 6 arguments. This means the first four arguments go into the fastcall registers i.e. RCX, RDX, R8 and R9. However, the remaining two arguments will have to be pushed to stack after allocating some homing space for our 4 registers. Make note that our top of the stack currently contains the return value for an internal NTAPI function TppWorkpExecuteCallback at Offset 0x130. This is how the callstack looks like when the callback function WorkCallback is called.

Call Stack								
Thread ID	D Address		То		From		Size	Comment
6400	000000000000000000000000000000000000000	ADEBD8	00007FFD19D72	260	0000000000040	)16E0	50	tpool.0000000004016E0
	000000000000000000000000000000000000000	ADFC28	00007FFD19D6	614	00007FFD19D7	2260 31AA	30	ntdil.jppworkerinread+68A
	000000000000000000000000000000000000000	ADFF58	00007FFD19D62	26A1 0000	00007FFD1974	17614 526A1	80	kerne 32.00007FFD19747614 ntd  .Rt UserThreadStart+21
00000000	00ADFBD8	00007	FFD19D72260	retu	rn to ntdl	l.Tpp	work	ExecuteCallback+130 from ???
00000000	00ADFBE0	00000	000000C5DE0					
00000000	00ADFBE8	00000	00000000000000					
00000000	00ADFBF0	00000	000000C5EA8					
00000000	00ADFBF8	00000	000000C0BC0					
00000000	00ADFC00	00000	0000000000000					
00000000	00ADFC08	00000	0000000000000					
00000000	00ADFC10	00000	000000000000					
00000000	00ADFC18	00000	00000000000					
00000000	00ADFC20	L00000	00000000000			_		
00000000	00ADFC28	00007	FFD19D631AA	retu	irn to ntdl	і.трр	Worke	erThread+68A from ???
00000000	00ADFC30	00000	000000000000000000000000000000000000000					
00000000	00ADFC38	00000	000000000000000000000000000000000000000					
00000000	00ADFC40	00000	000000000000000000000000000000000000000					
00000000	00ADFC48	00000						
00000000	OADECE	00000						
00000000	00ADEC60	00000	0010101010001					
00000000		000000	000000000000000000000000000000000000000					
00000000	00ADEC70	00000	010000000000					
00000000	00ADEC78	00000	000000000000000000000000000000000000000					
00000000	00ADFC80	00000	000000C0BC0					
00000000	00ADFC88	00000	000002D2000					
00000000	00ADFC90	00000	000000С0вс0					

Now heres the catch. If you modify the top of the stack where the return address lies, add the homing space for the 4 registers and add arguments to it, the whole stack frame will go for a toss and mess up stack unwinding. Thus we have to modify the stack without changing the stack frame itself, but by only changing the values within the stack frame. Each stack frame starts and ends at the blue line shown in the image above. Our stack frame for TppWorkpExecuteCallback has enough space within itself to hold 6 arguments. So our next step is to extract the data from our NTALLOCATEVIRTUALMEMORY\_ARGS structure and move it to the respective registers and stack. When we call TpAllocWork, we pass on the pointer to NTALLOCATEVIRTUALMEMORY\_ARGS structure is of 8 bytes (for x64, for x86 it would be in the RDX register now. Each value in our structure is of 8 bytes (for x64, for x86 it would be 4 bytes). So, we will extract these QWORD values from the structure and move it to RCX, RDX, R8, R9 and the remaining values on stack after adjusting the homing space. The calling convention for x64 functions in windows as per the msdn documentation would be:

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtAllocateVirtualMemory(
            HANDLE
                      ProcessHandle, // goes into rcx
  [in]
  [in, out] PVOID
                      *BaseAddress, // goes into rdx
 [in] ULONG_PTR ZeroBits, // goes into r8
[in, out] PSIZE_T RegionSize, // goes into r9
                      AllocationType, // goes to stack after adjusting homing space
 [in]
            ULONG
for 4 arguments
            ULONG Protect
                                       // goes to stack below the 5th argument after
  [in]
adjusting homing space for 4 arguments
);
```

Convering this logic to assembly would look like:

section .text

global WorkCallback

```
WorkCallback:
```

```
mov rax, [rbx]
                          ; backing up the struct as we are going to stomp rdx
                         ; NtAllocateVirtualMemory
mov rcx, [rbx + 0x8]
                        ; HANDLE ProcessHandle
mov rdx, [rbx + 0x10]
                         ; PVOID *BaseAddress
xor r8, r8
                         ; ULONG_PTR ZeroBits
mov r9, [rbx + 0x18]
                         ; PSIZE_T RegionSize
                        ; ULONG Protect
mov r10, [rbx + 0x20]
mov [rsp+0x30], r10
                        ; stack pointer for 6th arg
mov r10, 0x3000
                        ; ULONG AllocationType
mov [rsp+0x28], r10 ; stack pointer for 5th arg
jmp rax
```

To explain the above code:

- We first backup our pointer to the structure residing in the RDX register into the RBX register. We are doing this because we are going to stomp the RDX register with the second argument of NtAllocateVirtualMemory when we call it
- We move the first 8 bytes from the address in RBX register (struct NTALLOCATEVIRTUALMEMORY\_ARGS i.e UINT\_PTR pNtAllocateVirtualMemory) to rax register where we will jump to later after adjusting the arguments
- We move the second set of 8 bytes (HANDLE hProcess) from the structure to RCX
- We move the third set of 8 bytes i.e. pointer to a NULL pointer (PVOID\* address) stored in the structure into RDX. This is where our allocated address will be written by NtAllocateVirtualMemory
- We zero out the R8 register for the ULONG\_PTR ZeroBits argument

- We move the 6th argument i.e the last argument which should go to the bottom of all arguments (ULONG Protect i.e. PAGE permissions) to r10 and then move it to offset 0x30 from top of the stack pointer.
  - Top of the stack pointer = RSP = Return address of TppWorkpExecuteCallback which is 8 bytes
  - Homing space size for 4 arguments = 4x8 = 32 bytes
  - Space for the 5th argument = 8 bytes
  - Thus 32+8 = 40 = 0x28 (this is where the second last 5th argument will go)
  - Thus 32+8+8 = 48 = 0x30 (this is where the last 6th argument will go)
- We finally move the 5th argument value (ULONG AllocationType) i.e. 0x3000 MEM\_COMMIT | MEM\_RESERVE to the R10 register and then push it to offset 0x28 from the RSP

Compiling it all together, this is what it looks like before jumping to NtAllocateVirtualMemory:

- The disassembled code shows the asm instructions we wrote. The current instruction pointer is just after adjusting the stack and before jumping to NtAllocateVirtualMemory
- The registers show the arguments for NtAllocateVirtualMemory
- The Dump shows the NTALLOCATEVIRTUALMEMORY\_ARGS structure in memory. Each 8 byte memory block is an object relating to the contents of the strucutre
- The stack shows the adjusted stack for NtAllocateVirtualMemory

CPU De Log Notes • Breakpoints	🛲 Memory Map 🛛 🧟 SEH 🛛 👩 Script 🛛 🎴 Symbols	🛇 Source 🔎 References 👒 Threads 着	Handles 👔 Trace 📋 Call Stack					
0000000004016E0 0000000004016E3	48:89D3 mov rbx,rdx 48:8803 mov rax,gword ptr ds:[rb	rax:ZwAllocateVirtualMemory	^	Hide FPU				
0000000004015E6 00000000004015EA 00000000004015E1 00000000004015F1 00000000004015F5 00000000004015F9 0000000004015F9	48:88340         08         mov rcx,qword ptr ds:[rb           48:885310         mov rcx,qword ptr ds:[rb           49:885310         mov rcx,qword ptr ds:[rb           40:31c0         xor r8,r8           40:88542         mov r10,qword ptr ds:[rb           40:885320         mov r10,qword ptr ds:[rb           40:88542430         mov r10,qword ptr ds:[rb           40:89542424         mov r10,qword ptr ds:[rb           40:89542428         mov r10,qword ptr ds:[rb+28	xx+8] xx+10] xx+18] xx+20] ASM-Ecode 3],r10	RAX         00007FFD19DAD3B0           RBX         0000000064FpD0           RCX         0000000064FpD0           RCX         00000000064Fp00           RBP         000000000756468           RSP         000000000075650           RSI         0000000075650	<pre><ntdll.zwallocatevirtualmemory></ntdll.zwallocatevirtualmemory></pre>				
RIP         000000000000000000000000000000000000	Jmp rax         Jmp rax           0F1F4400 00         dword ptr ds:[rax+ra           48:8810         sub rsp.28           48:8805 F5180000         mov rax gword ptr ds:[fax+ra           48:8800         rax gword ptr ds:[fax+ra	AX],eax 33010] Tax:ZWAllocateVirtualMemory Pay:ZWAllocateVirtualMemory Pay:ZWAllocateVirtualMemory	RDI 000000007FFE0386 R8 00000000000000000 R9 00000000064FDF8 R10 000000007FFE0008 R11 00000007FFE0008 R12 00000007FFE0088	→ ULONG_PTR ZeroBits → PSIZE_T RegionSize				
.text:0000000000001278         text:000000000000022c         L'O'           NTALLOCATEVIRTUALMEMORY_ARGS struct Memory         R13         00000000000022c         L'O'								
🚛 Dump 1 🚛 Dump 2 🚛 Dump 3 🚛 D	ump 4 Dump 5 🛞 Watch 1 🛛 Locals 🎾 Struct	000000000C4FBD8 00007	FFD19D72260 return to ntdl	<pre>1.TppWorkpExecuteCallback+130 from ???</pre>				
Address Hex 00000000064FDD0 B0 D3 DA 19 FD 7F 000000000064FDE0 00 FE 64 00 00 00	ASCII 00 00 FF FF FF FF FF FF FF <sup>*</sup> 0 <sup>(1</sup> , <sup>(1</sup> ), <sup>(1</sup> ), <sup>(1</sup> ), <sup>(1</sup> )	00000000000000000000000000000000000000	000007563A0 0000000000 00000756468 00000750BC0	4 registers				
00000000064FDF0 20 00 00 00 00 00 00000000064FE00 00 00 00 00 00 00 00000000064FE10 <u>C0 28 D2 19 FD 7F</u>	00         00         10         00<	000000000000000000000000000000000000	00000003000 +ULONG Allocation 00000000020 + ULONG Protect - P 00000000000	Ivpe - MEM_COMMITIMEM_RESERVE AGE_EXECUTE_READ				
00000000064FE20 00 00 00 00 00 00 00 00 00 00 00 00 0	00 00 B4 13 40 00 00 00 00 00 00	00000000000000000000000000000000000000	000000000000000000000000000000000000000					

And a quick look at the stack after the execute of NtAllocateVirtualMemory shows a valid callstack which can be unwinded perfectly. You can also see that the syscall for NtAllocateVirtualMemory returned zero which means the call was successful.

😑 😏 🖬 🔶 🕸 🙀 😓 🙀 😓 🕴 🖉 🕫 📓 🥜 层 🕢 🦧 fx #   Az 🏥 🖥 🖤								
🔛 CPU 🛛 📝 Log	Notes   Breakpoints	🛲 Memory Map 🛛 🧧 SEH 🛛 👩 Script 🔮 Symbols 🔇 Source	🔎 References 🛛 🛸 Threads 🛛 🔒 Ha	indles 👔 Trace 📋 Call Stack				
:	00007FFD19DAD3B0 <	4C:8BD1 mov r10,rcx B8 18000000 mov eax.18	rcx:NtAllocateVirtualMemory		Hide FPU			
RIP RCX	00007FFD19DAD3B8 00007FFD19DAD3C0 00007FFD19DAD3C2 00007FFD19DAD3C4	F60425 0803FE7F (test byte ptr ds:[7FFE0308],1 75 03 0F05 C3 		RAX         00000000000000           RBX         00000000064FDD0           RCX         00007FD19DAD3c4           RDX         000000000000           RBP         00000000000756468	<&zwAlllocateVirtualMemory> ntdll.00007FFD19DAD3C4 <&TppWorkpTaskVFuncs>			
	General Statistics Performance Thr	reads Token Modules Memory Environment Handles GPU Comment	cx:NtAllocateVirtualMemory	RSP 000000000004FBD8 RSI 0000000007565C0 RDI 000000007FFE0386				
	TID         CPU         Cycles delt           1300         8208         7704         357,67           73512         3572         3572         3572	to         Start address         Priority           tpool.exe+0x140         Normal           rtdldlTpReleaseCleanupGroupMembers+0x450         Normal           Y4         ntdldlTpReleaseCleanupGroupMembers+0x450         Normal           rtdldlTpReleaseCleanupGroupMembers+0x450         Normal		R8         000000000C4FBD8           R9         000000000756468           R10         00000000000000           R11         00000000000346           R12         0000000007FFE03B0	<&TppWorkpTaskVFuncs> L'^			
.text:00007FFD19D		Stack - thread 7704	×	R13 00000000000022C R14 000000000C4FDA8	L'ō'			
Will Dump 1         Will           Address         0000000064F           00000000064F         00000000064F           000000000064F         00000000064F           000000000064F         00000000064F           00000000064F         00000000064F           00000000064F         00000000064F           00000000064F         0000000064F           00000000064F         0000000064F	Start C:\Windows\Syster Startet: 1:24:50 AM 1	Name         O           0         ndtil.dllNAllocateVirtual/Nemory+0x14           1         ndtil.dllTpAlloche0+0xa60           2         ndtil.dllTpAlloche0+10xa6           3         kernel32.dllBaseThreadIthreadThum+014           4         ndtil.dllRtlUserThreadStart+0x21	D00000000C4FBE3         0000000           000000000C4FBE3         0000000           000000000C4FBE3         0000000           000000000C4FBE3         0000000           000000000C4FBE3         0000000           000000000C4FFE4         0000000           000000000C4FFE4         0000000           000000000C4FFE3         0000000           000000000C4FFE3         0000000           000000000C4FFE3         0000000           000000000C4FE3         0000000           000000000C4FE3         0000000           000000000000000000000000000000000000	D19072260 return to ntdl1 0007563A0 00007563A0 000056468 000000000 000000000 000000000 00000000	.TppWorkpExecuteCallback+130 from ??? .TppWorkerThread+68A from ???			
000000000064F 000000000064F 00000000064F 00000000064F	State:         Wait:Executive           Kernel         00:00:00.015           User time:         00:00:00.000           Context switches:         18,040		00000000000000000000000000000000000000	0007508C0 000000000 0007508C0 00007508C0				

The stack is as clear as crystal again with no signs of anything malevolent. Make note that this is not stacking spooing, because in our case the stack is being unwinded fully without crashing. There are many more such API calls which can be used for proxying various functions; which I will leave it out to the readers to use their own creativity. The upcoming release of BRc4 will use something similar but with different set of API calls which are fully undocumented and will be under a different payload option called as stealth++. The full code for this can be found in my github repository.

Tagged with: <u>red-team blogs</u>