Malware development on secured environment

Write, Adapt, Overcome

08/11/2023

/ 01 Whoami

WHOAMI

And why should we trust you ?

Muggle identity

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

Experience

- Senior pentester @*Wavestone* for almost 4 years
- Dedicated to large-scale *RedTeam* operations – *CAC40* companies
- Development of internal tooling – Mainly malware and Cobalt
- Uncommon process injection pattern – @LeHack 2023

/ **02**

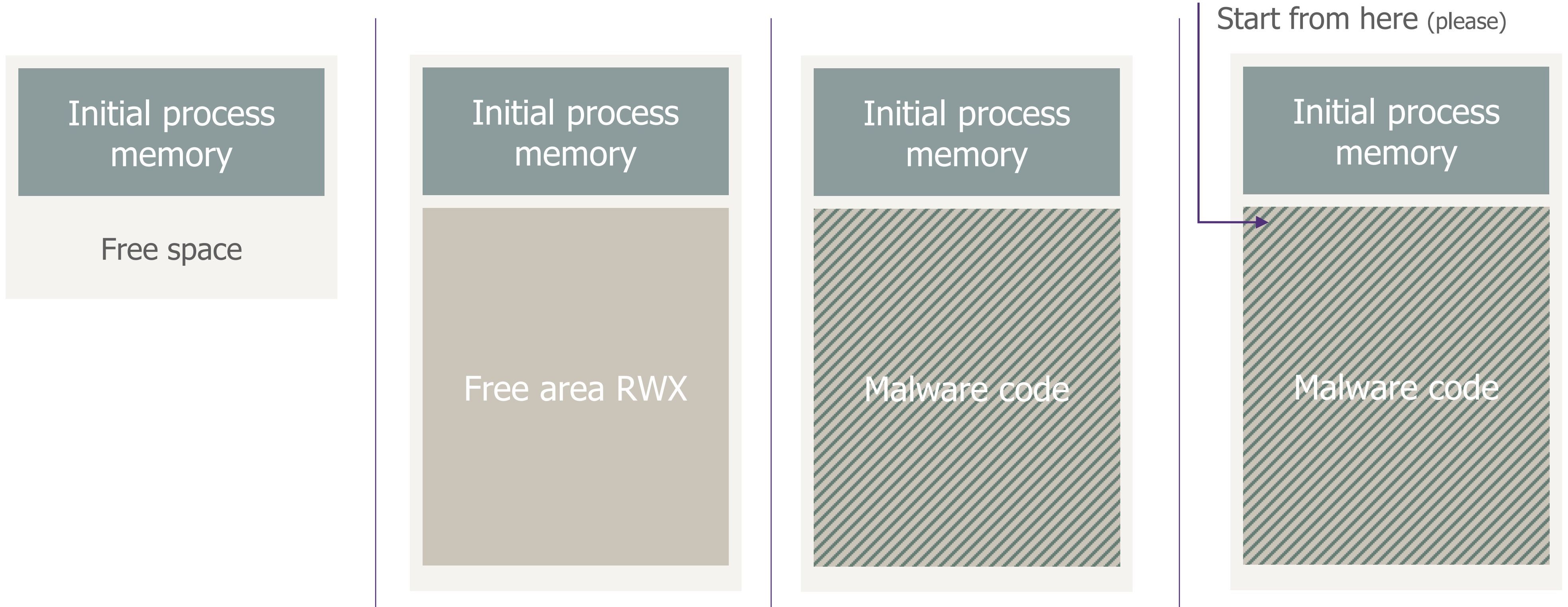
First injection

You always remember your first one

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



VirtualAllocEx
VirtualProtectEx

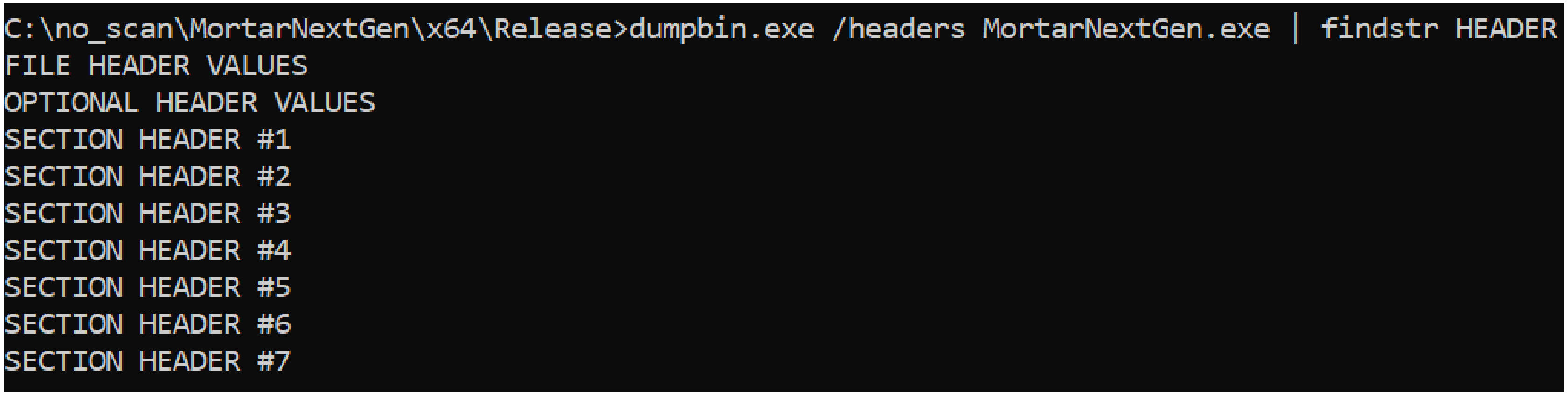
WriteProcessMemory

CreateRemoteThread

Store the payload

What about PE format

- PE : Portable Executable
- PE are organized in headers and sections



```
C:\no_scan\MortarNextGen\x64\Release>dumpbin.exe /headers MortarNextGen.exe | findstr HEADER
FILE HEADER VALUES
OPTIONAL HEADER VALUES
SECTION HEADER #1
SECTION HEADER #2
SECTION HEADER #3
SECTION HEADER #4
SECTION HEADER #5
SECTION HEADER #6
SECTION HEADER #7
```

Interesting sections

- **`.text` : executable code**
- `.rdata` : read-only data
- **`.data` : global initialized variables**
- `.pdata` : exception information
- **`.rsrc` : files embedded in the executable**
- `.reloc` : used to handle base address offset (will not be seen today)

Store the payload

Store the payload in the sections

- The payload can be stored in any section
- It is usually stored in the `.text`, `.data`, `.rdata` or `.rsrc` section

.text

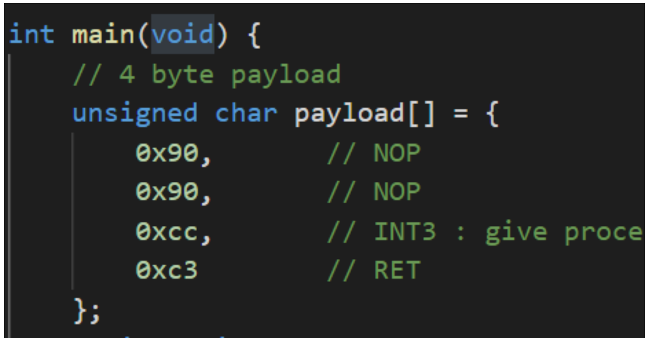
- The payload is directly stored in a function
- Harder to modify on-the-fly because it need modification of the code and compilation can fail due to the payload size

.data

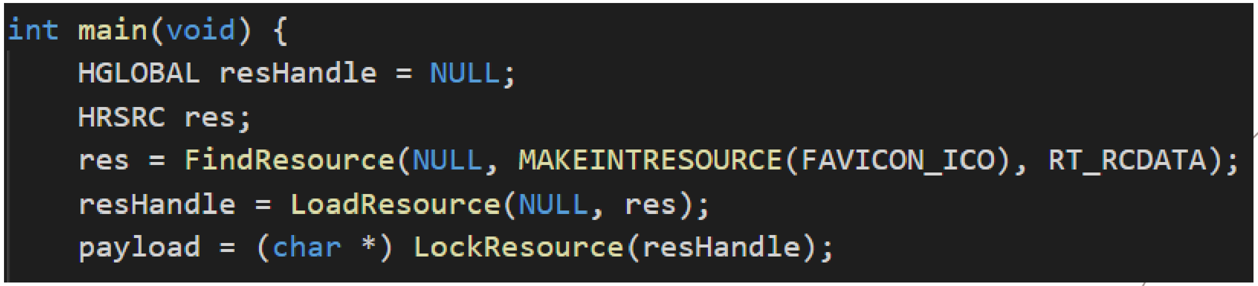
- The payload is stored in a global variable
- Can take time at compile time but and the payload must be pre-processed

.rsrc

- The payload is stored as a resource (`.txt` file for example)
- The payload is stored in a simple file and linked to the PE by the linker



```
int main(void) {
    // 4 byte payload
    unsigned char payload[] = {
        0x90,         // NOP
        0x90,         // NOP
        0xcc,         // INT3 : give proce
        0xc3          // RET
    };
```



```
int main(void) {
    HGLOBAL resHandle = NULL;
    HRSRC res;
    res = FindResource(NULL, MAKEINTRESOURCE(FAVICON_ICO), RT_RCDATA);
    resHandle = LoadResource(NULL, res);
    payload = (char *) LockResource(resHandle);
```

Hands on
> First process injection

Retrieve the payload

- Make a function that will retrieve a payload stored in the `.data` section
- Update the function to retrieve the payload encoded in base64
- Update the function to retrieve the payload xored with a static key and encoded in base64

Perform your first injection

- Open the remote process with `OpenProcess`
- Allocate some writable memory in the process using `VirtualAllocEx`
- Write your malicious payload in memory using `WriteProcessMemory`
- Re-protect the memory with `RX` rights using `VirtualProtectEx`
- Run the payload in a new thread with `CreateRemoteThread`

Additional steps

- Try to hide the different imports by using `GetProcAddress`
- Try to implement some basic entropy bypass

FIRST INJECTION

CheatSheet
> Covenant

Run Covenant

- Run `dotnet run` in the **Covenant** directory
- Go to `https://<ip>:7443`

Create a listener

- The listener is the service that will handle beacon connections
- `Listeners` > `Create` > `Create`

Create a beacon

- `Launcher` > `ShellCode`
- Set `DotNetVersion` to `Net40`
- `Generate` then `Download`

CheatSheet

> MDE

Global alerts

- https://security.microsoft.com/alerts
- This tab shows alerts triggered by **MDE**
- I didn’t implement specific rules on **MDE** so the alerts may not be all reported here

Detailed telemetry

- `Devices > Machine > Timeline`
- Contains all the telemetry raised by the device
- You will be able to easily track actions performed by your binary and the events raised by **MDE**
- Try to use it as much as possible (the data can take up to 10 minutes to come)

Hands on
> *First process injection - Analysis*

Retrieve the payload

- Using **PE-Bear**, check the different section sizes. Try to hide the payload in `.data` and `.rdata`
- Do you see any difference with a plain payload and a xored one?

Perform your first injection

- At each steps, check the remote process memory state using **ProcessHacker**
- Once the thread has been created in the remote process, check the thread’s stack using **ProcessHacker**.
- What IOC could you find to detect such injection?
- Run it against **MDE**, what are the different alerts and why are they raised?

Additional steps: hide the imports

- Check your imports with **dumpbin**
- If you look at the binary’s strings, can you still find your function’s name?

Process injection 101
> Standard pattern

Main idea

- Modify the memor
- Compel the injecte

Initial process memory

Free space

t from here (please)

Initial process memory

Malware code

eateRemoteThread

VirtualProtectEx

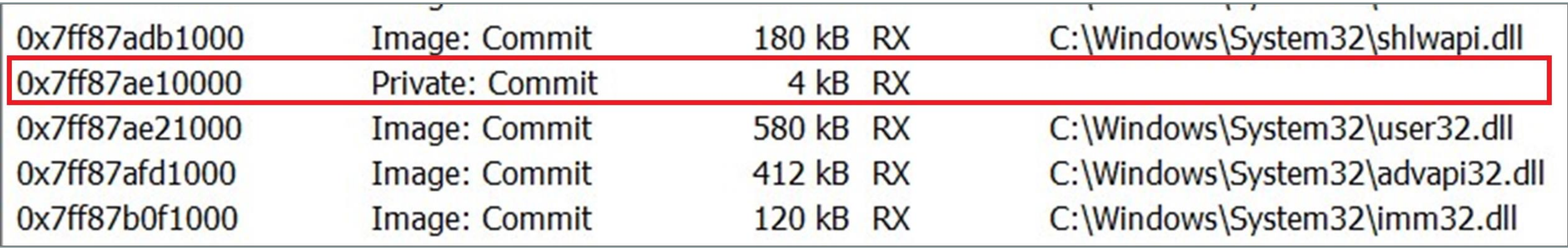
/ 03 Advanced process injection methods

DLL Injection and Module Stomping

Allocation primitives: VirtualAllocEx
> File backed and unbacked memory

Effect of VirtualAllocEx

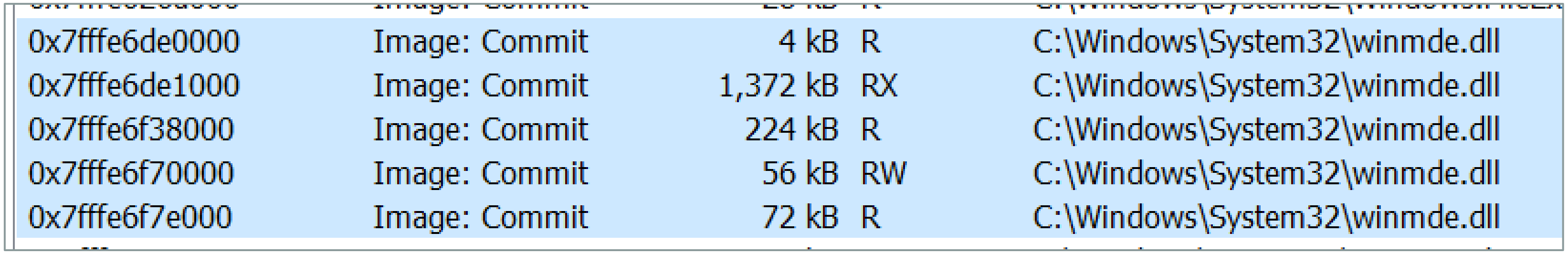
- The allocated memory space is not recognized to have any use by the system
- Execution from unbacked memory could raise some low levels alerts. It is quite unusual to execute code from an unbacked memory even if some binary such as C# one heavily use it.



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 a file
- The memory space is known to have a real purpose

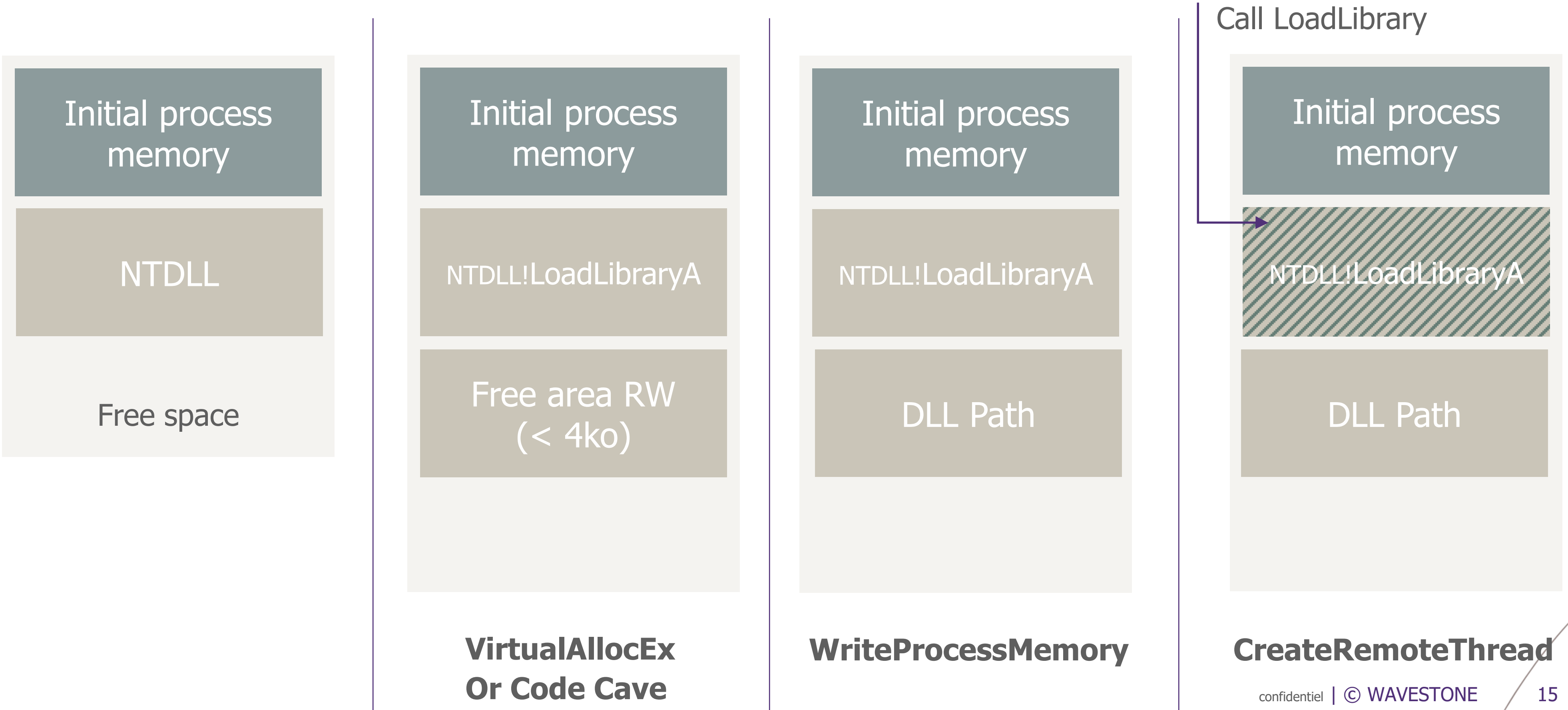


0x7fffe6de0000	Image: Commit	4 kB	R	C:\Windows\System32\winmde.dll
0x7fffe6de1000	Image: Commit	1,372 kB	RX	C:\Windows\System32\winmde.dll
0x7fffe6f38000	Image: Commit	224 kB	R	C:\Windows\System32\winmde.dll
0x7fffe6f70000	Image: Commit	56 kB	RW	C:\Windows\System32\winmde.dll
0x7fffe6f7e000	Image: Commit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

How to do it ?
> Use VirtualAllocEx to avoid VirtualAllocEx

VirtualAllocEx again ?

› Some EDR (S1, MDE, Sophos) does not seem to be bothered by allocation of less than 4ko



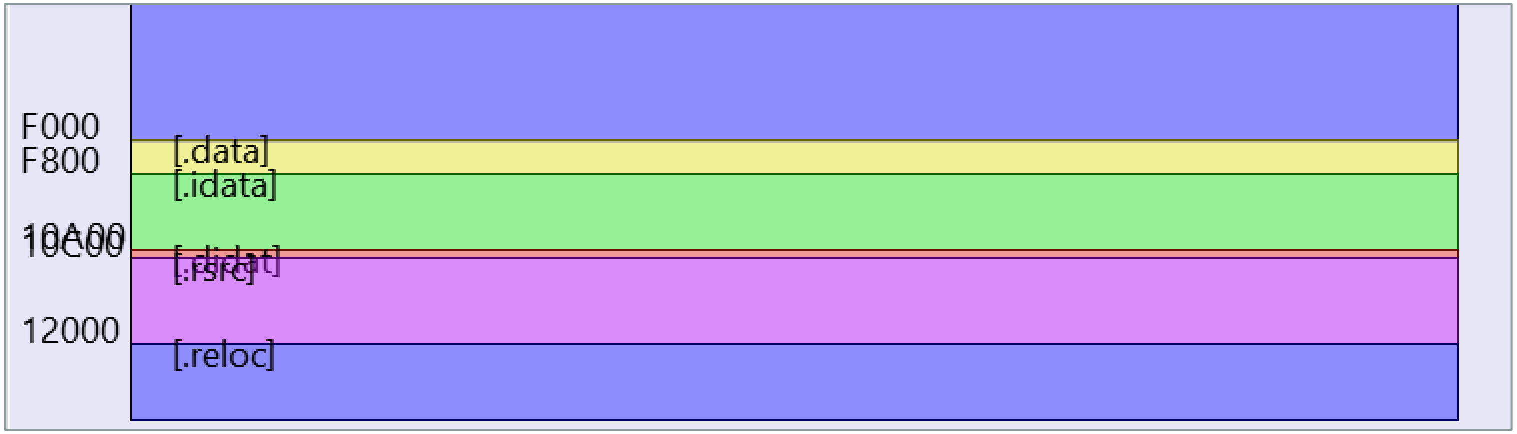
What’s next with it ?
> Limit the use of VirtualProtect by reusing DLL sections

Reuse the DLL sections …

- DLL have predefined sections with specific ReadWriteExecute (RWX) rights
- It is interesting to write your malware on the DLL’s `.text` section

… And be careful

- 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



Hands on
> *Module Stomping*

Retrieve the payload

- Just use one of your previous function!

Perform the self injection

- Load the library `winmde.dll` in the process using `LoadLibraryA`
- Check the DLL with **PE-Bear** and choose an interesting function
- Resolve the function's address using `GetProcAddress`
- Write your payload at the function's address using `VirtualProtect` and `WriteProcessMemory`
- Call the function !

Hands on
> *Module Stomping - Analysis*

Perform the self injection

- Check that the DLL is well injected with **ProcessHacker**
- Check that the chosen function is well overwritten
- What about the memory section where the payload has been written? Is it a backed memory?

Run against MDE

- Check the malware against **MDE**
- Does it raise any alerts about malicious memory allocation?
- Does the method involve new thread creation? Why?

Hands on
> *DLL injection*

Retrieve the payload

- Just use one of your previous function!

Perform the injection

- Open the remote process
- Inject the DLL into the remote process
- Retrieve the DLL Base address
- Retrieve the function that will be stomped
- Stomp the function with the malicious code
- Run the malicious code with `CreateRemoteThread`

Hands on

> *DLL Injection - Analysis*

Perform the injection

- Check that the DLL is well injected with **ProcessHacker**
- Check that the chosen function is well overwritten
- What about the memory section where the payload has been written? Is it a backed memory?

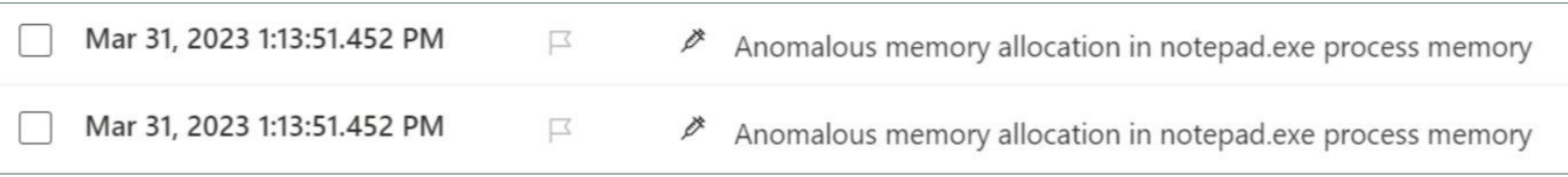
Run against MDE

- Check the malware against **MDE**
- Does it raise any alerts about malicious memory allocation?
- Does the method involve new thread creation? Why?

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
```

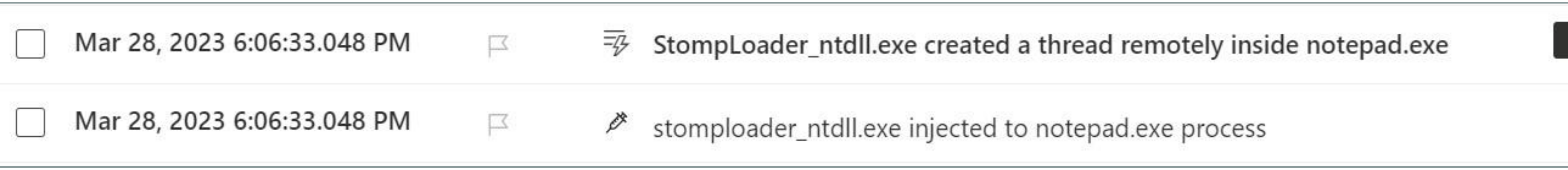
Detection with Module Stomping

- 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 ?

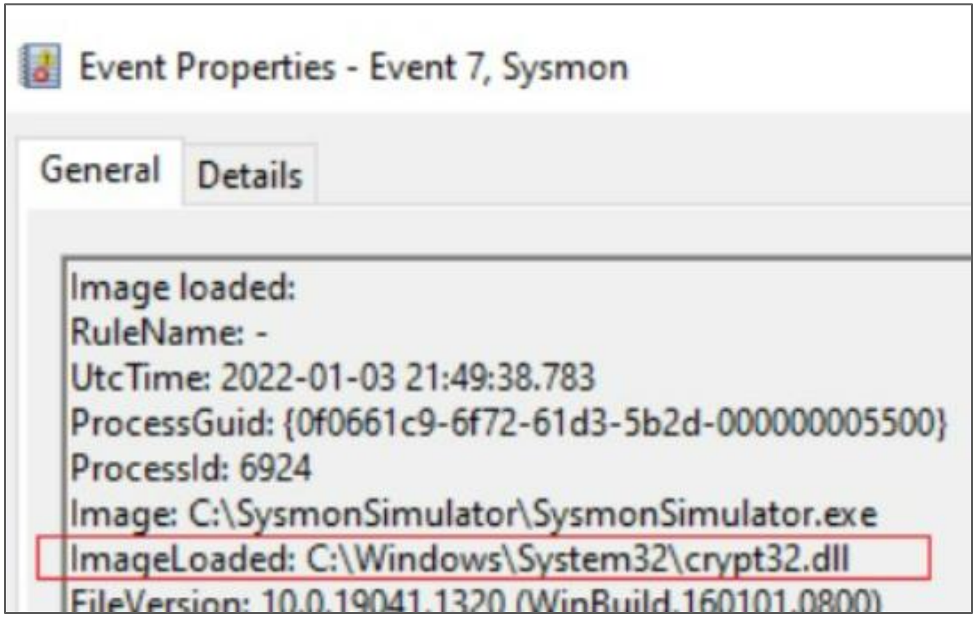
IOC

- `LoadLibraryA` still raises an *ETW* event that can be caught by security solutions
- 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)
```

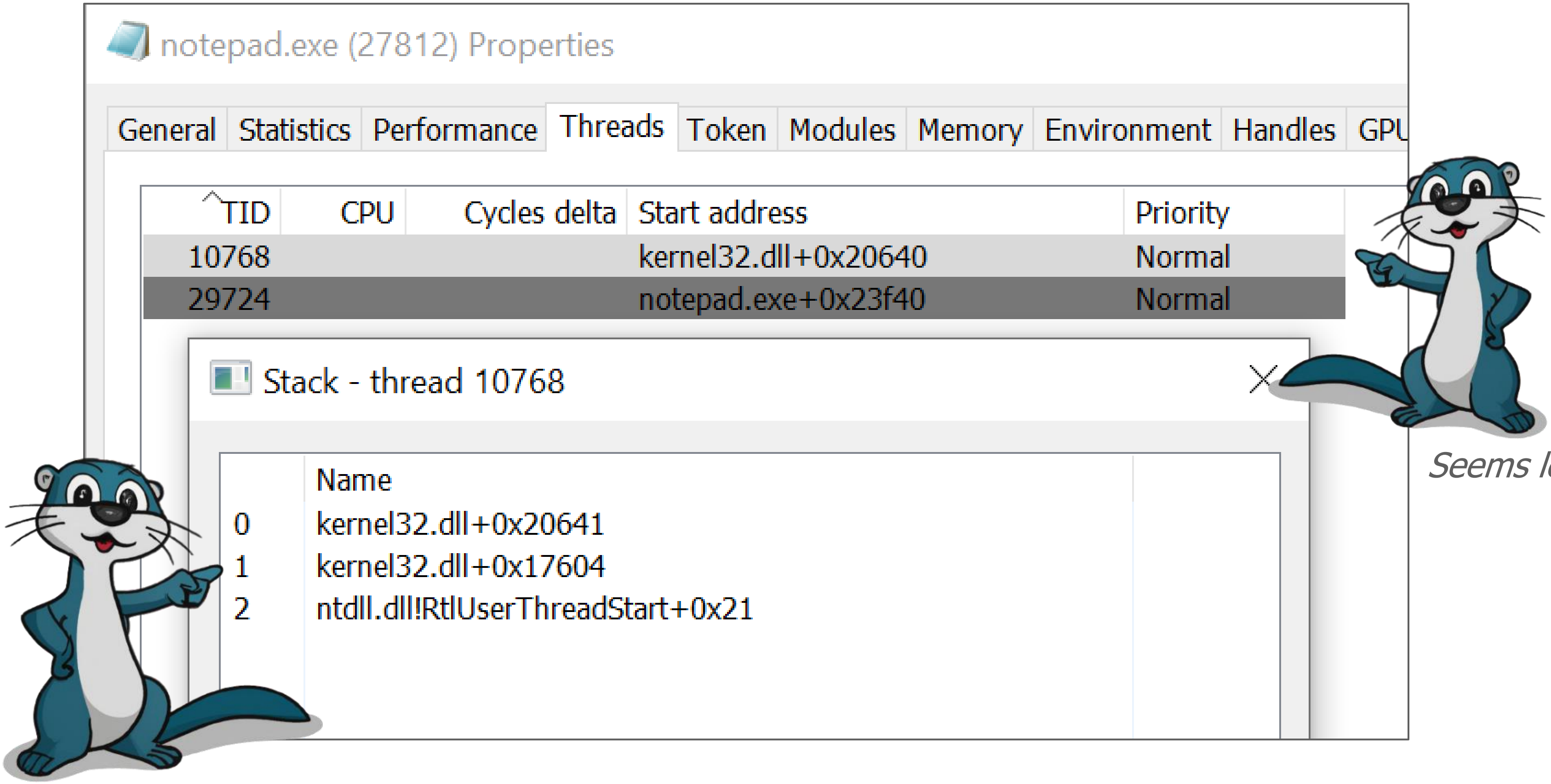
Sysmon catches the kernel event raised by the use of LoadLibrary and generates the related event on the Windows EVTX

/ 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
- Creation of an additional thread in a well known process can be used as an IOC as this is an unusual behavior



notepad.exe (27812) Properties

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

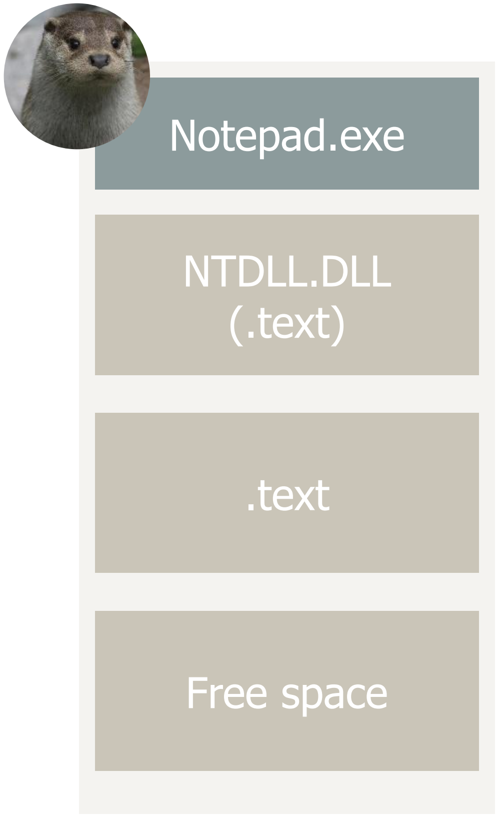
- The goal is to compel the program to execute a given code
- Instead of relying on the `CreateRemoteThread`, we will just wait for the injected process to run the malicious code

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



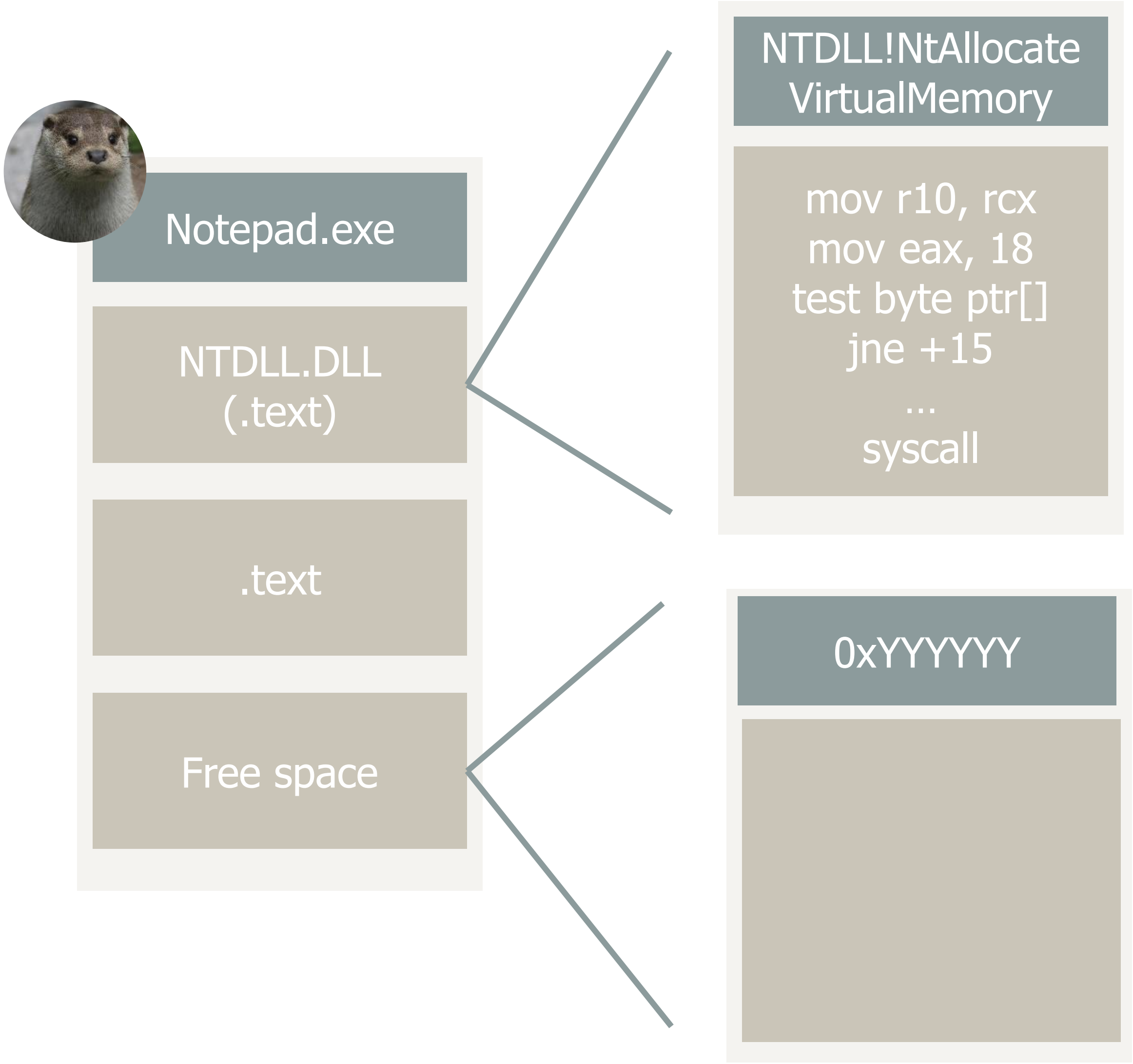
Notepad.exe

NTDLL.DLL
(.text)

.text

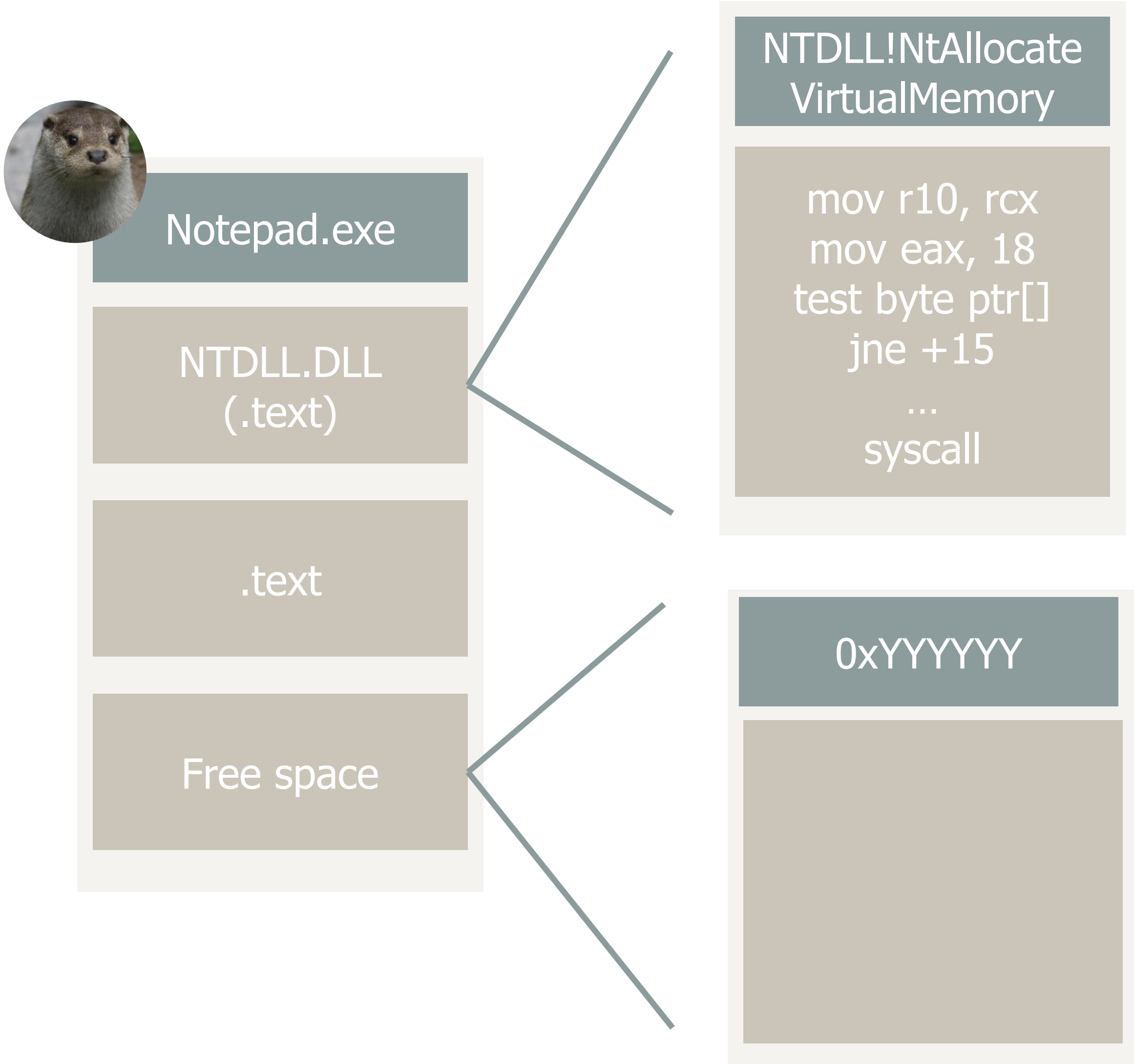
Free space

Execution primitives: *CreateRemoteThread*
> A little push up – API Hooking



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 – API Hooking



Notepad.exe

NTDLL.DLL
(.text)

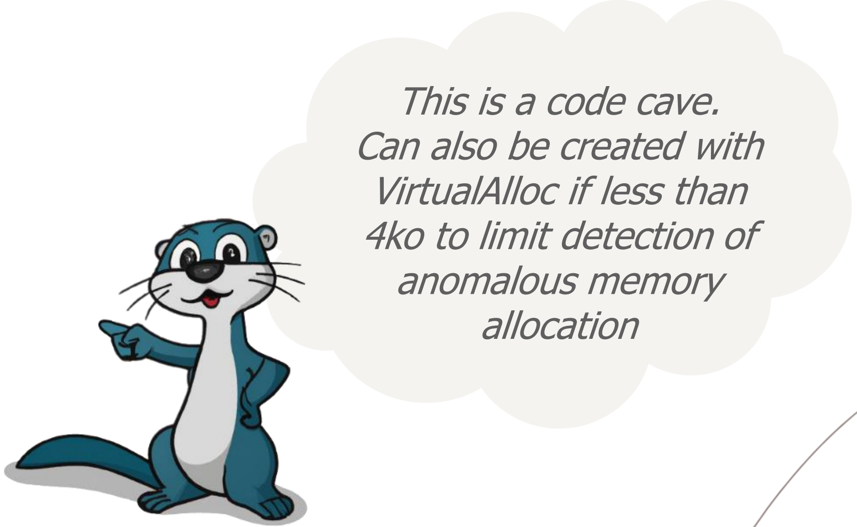
.text

Free space

NTDLL!NtAllocate
VirtualMemory

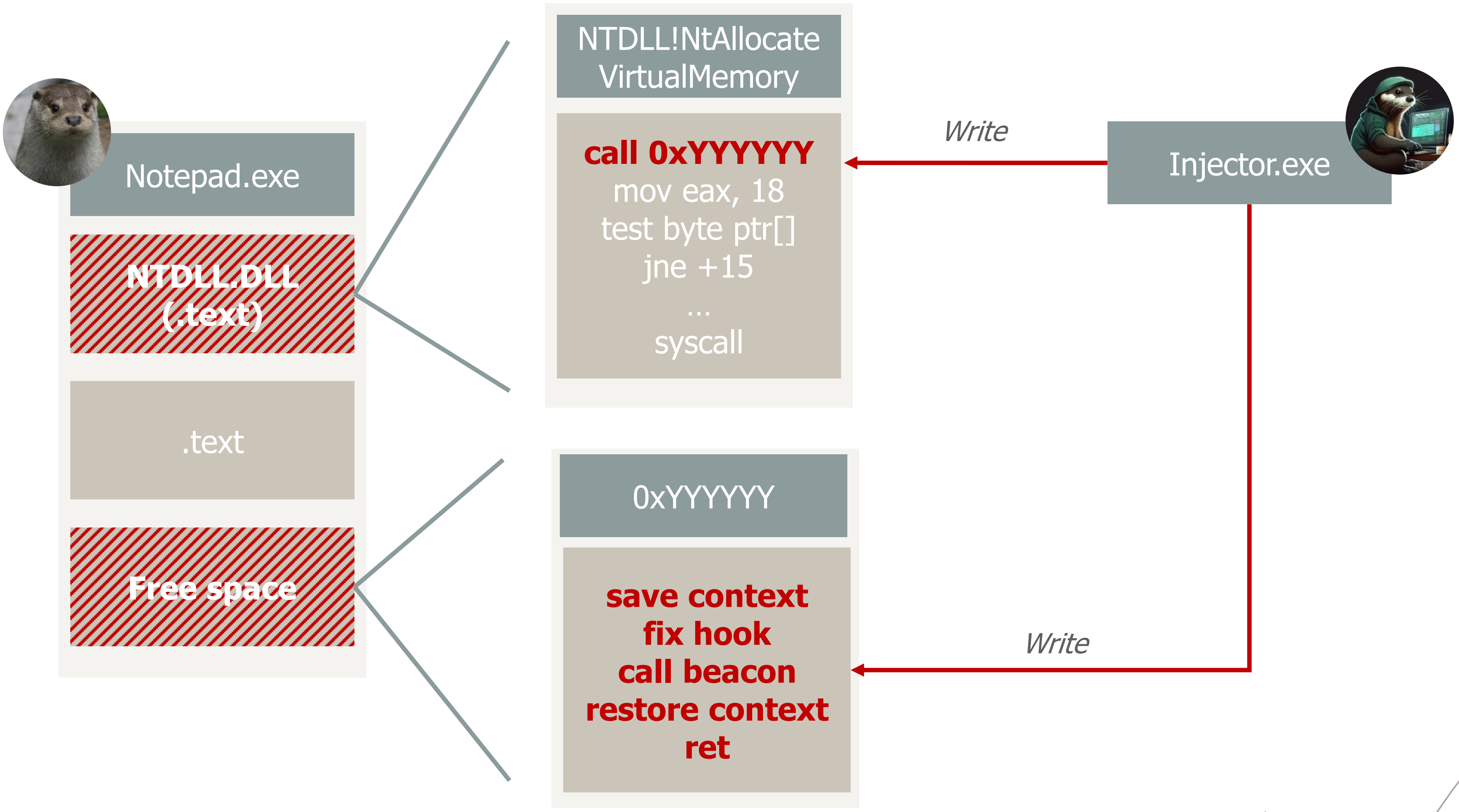
```
mov r10, rcx
mov eax, 18
test byte ptr[]
jne +15
…
syscall
```

0xYYYYYY



*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 – Redirect execution flow



Notepad.exe

NTDLL.DLL (.text)

.text

Free space

NTDLL!NtAllocateVirtualMemory

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

0xYYYYYY

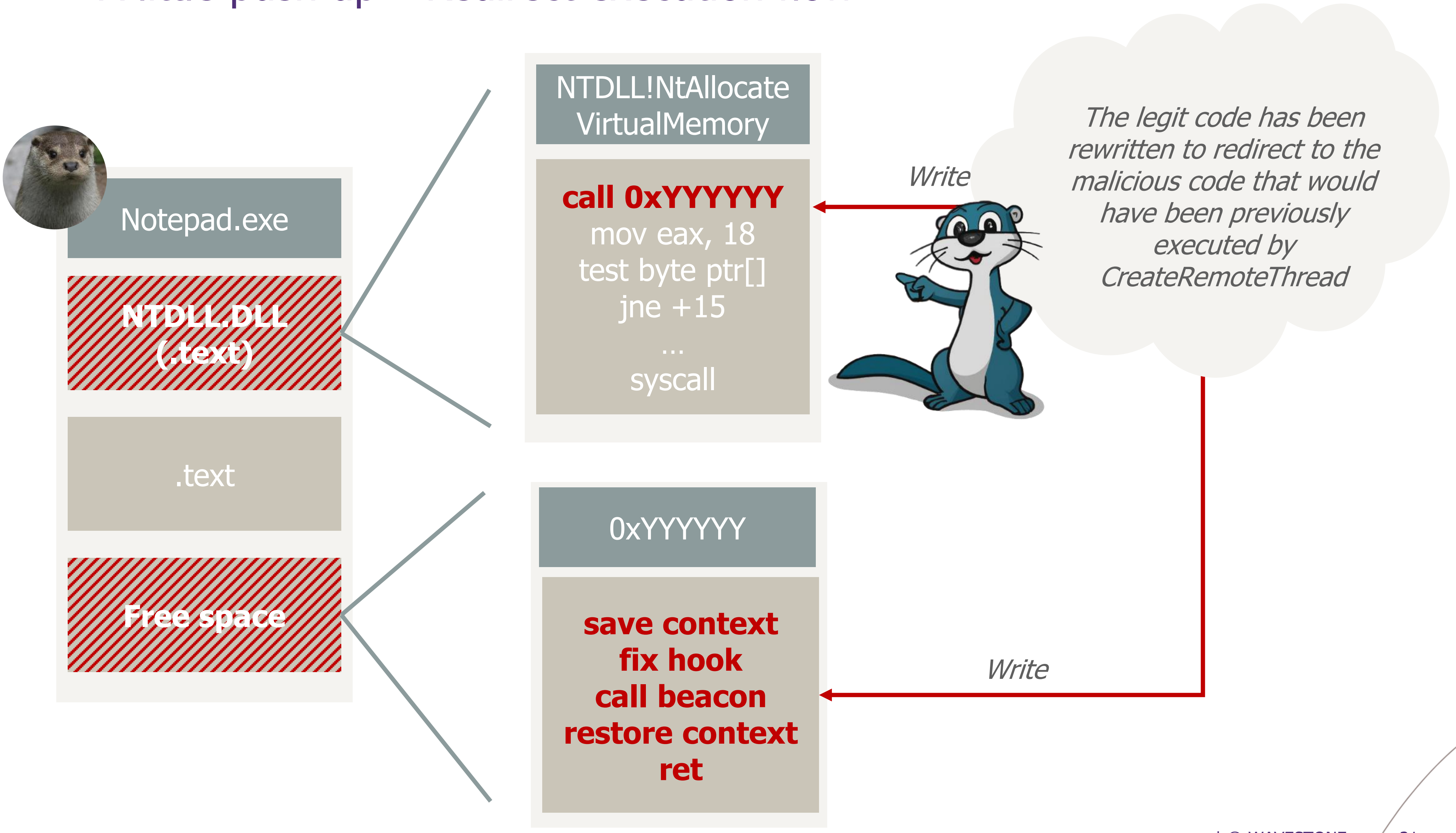
save context
fix hook
call beacon
restore context
ret

Write

Injector.exe

Write

Execution primitives: *CreateRemoteThread*
> A little push up – Redirect execution flow



Notepad.exe

NTDLL.DLL
(.text)

.text

Free space

NTDLL!NtAllocate
VirtualMemory

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

0xYYYYYY

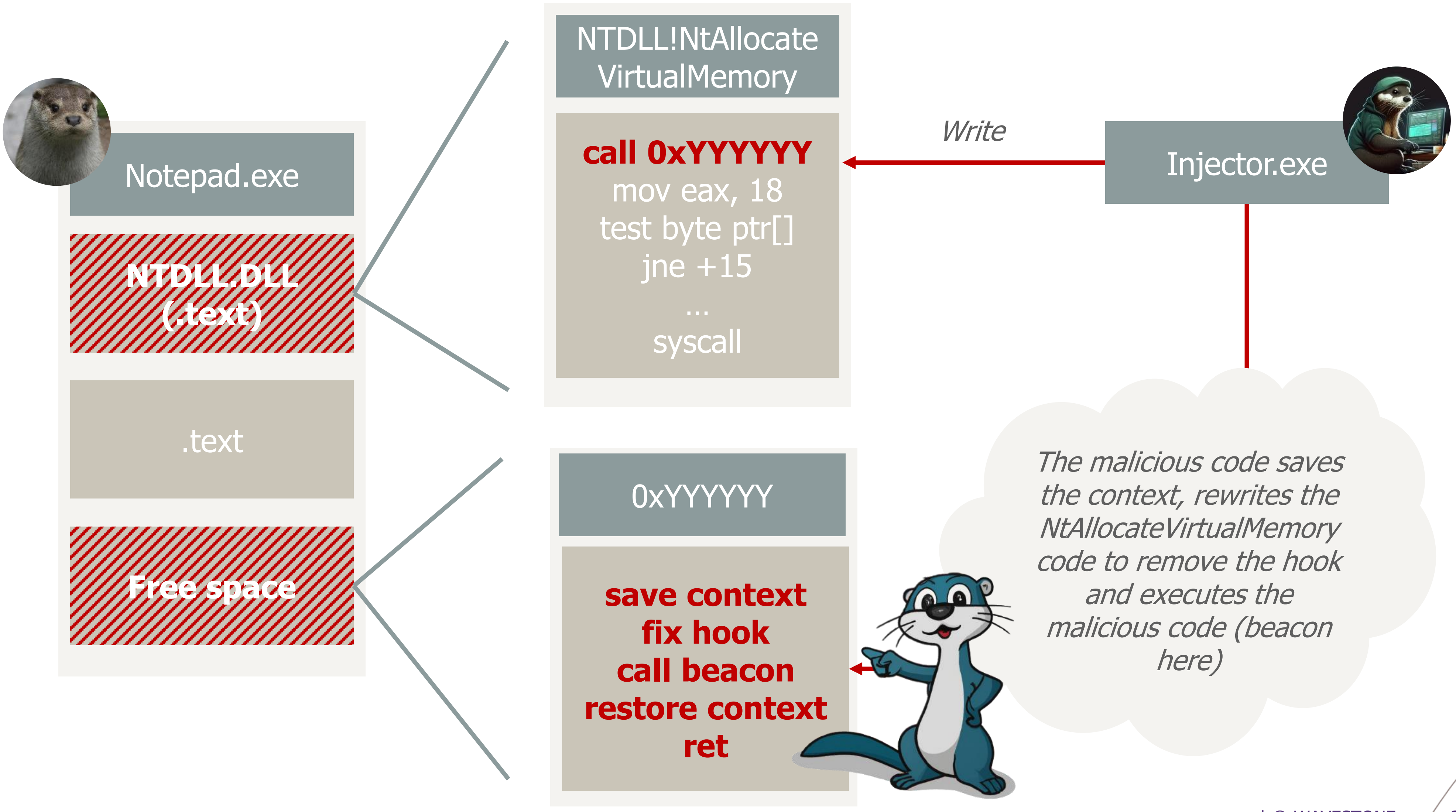
save context
fix hook
call beacon
restore context
ret

Write

Write

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

Execution primitives: *CreateRemoteThread*
> A little push up – Redirect execution flow



Notepad.exe

NTDLL.DLL (.text)

.text

Free space

NTDLL!NtAllocateVirtualMemory

```
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)

Hands on

> *Threadless injection*

Perform a basic DLL injection

- Just use one of your previous function!

Perform the threadless injection

- Modify the `CreateRemoteThread` used for the remote DLL injection
- Create an ASM code that will call `LoadLibrary`
- Create the ASM code that will be used as a hook
- Create the ASM code that will be used to save the context, rewrite the hook and call the malicious code
- Put it all together…

Hands on
> *Threadless Injection - Analysis*

Perform the injection

- Set a breakpoint on the hooked function
- Use the debugger to follow the execution flow
- Enjoy seeing the execution flow rerouted by your hooks

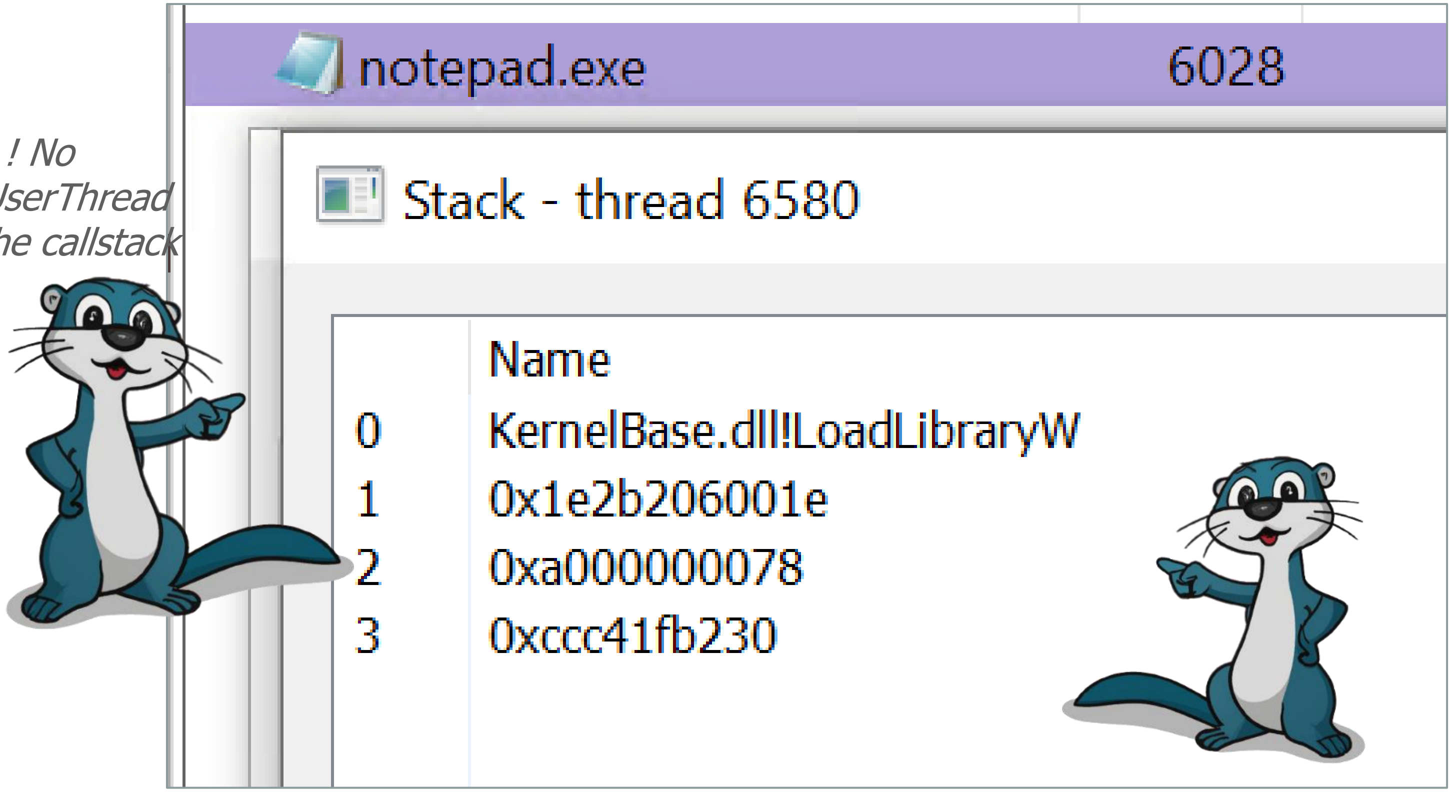
Run against MDE

- Check the malware against **MDE**
- Does it raise any alerts about malicious memory allocation?
- Does the method involve new thread creation?
- Does it raise any alerts about malicious thread creation?
- What about malicious memory protection?

Execution primitives: ThreadLess injection > Thread and threadless

Effect of the ThreadLess injection

- The malicious code has been successfully executed without using `CreateRemoteThread`
- The injection does not modify too much the standard process behavior, limiting the creation of small signals



notepad.exe 6028

Stack - thread 6580

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

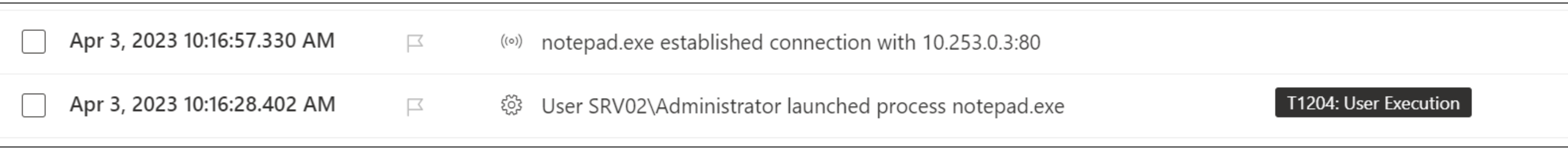
Yey ! No RtlUserThread in the callstack

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



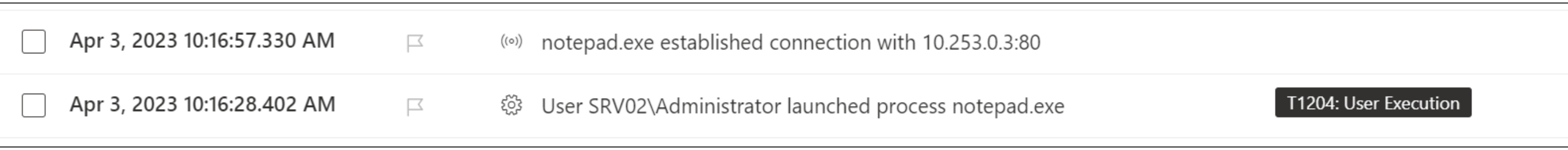
□ Apr 3, 2023 10:16:57.330 AM notepad.exe established connection with 10.253.0.3:80

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

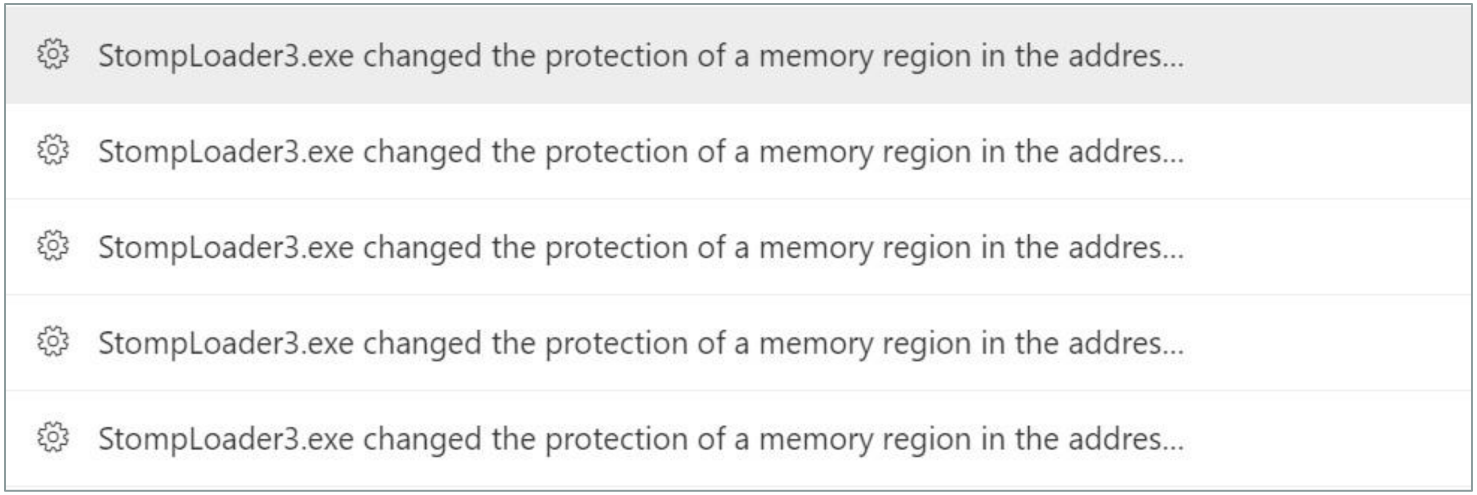
Synthesis (1/2)
> What does an EDR say about it ?

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...
```

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 thread stack 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

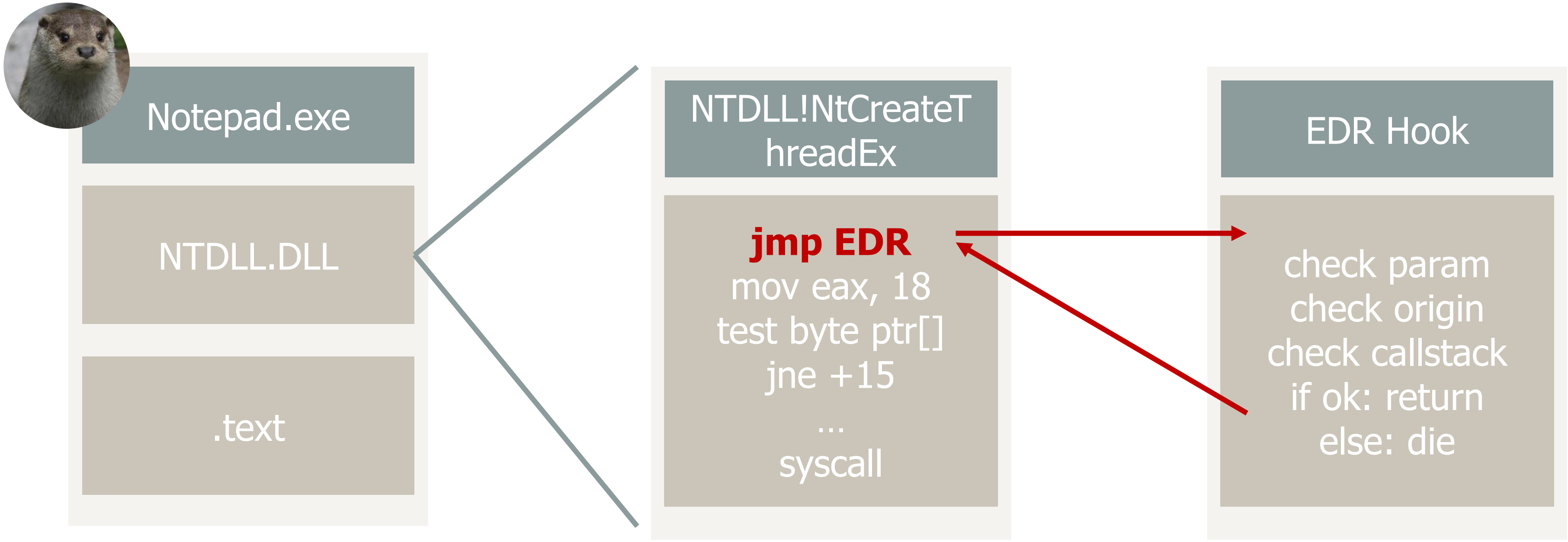
- The injection is still sensible to *EDR* hooks
- The injector can still be flagged as malicious once the injection ended
- Bypassing EDR hooks can be a nice addition

/ 05 Nothing to see here : bypassing userland hooks

EDR hooks 101
> Hooks, Userland and KernelLand

Interest of EDR hooks

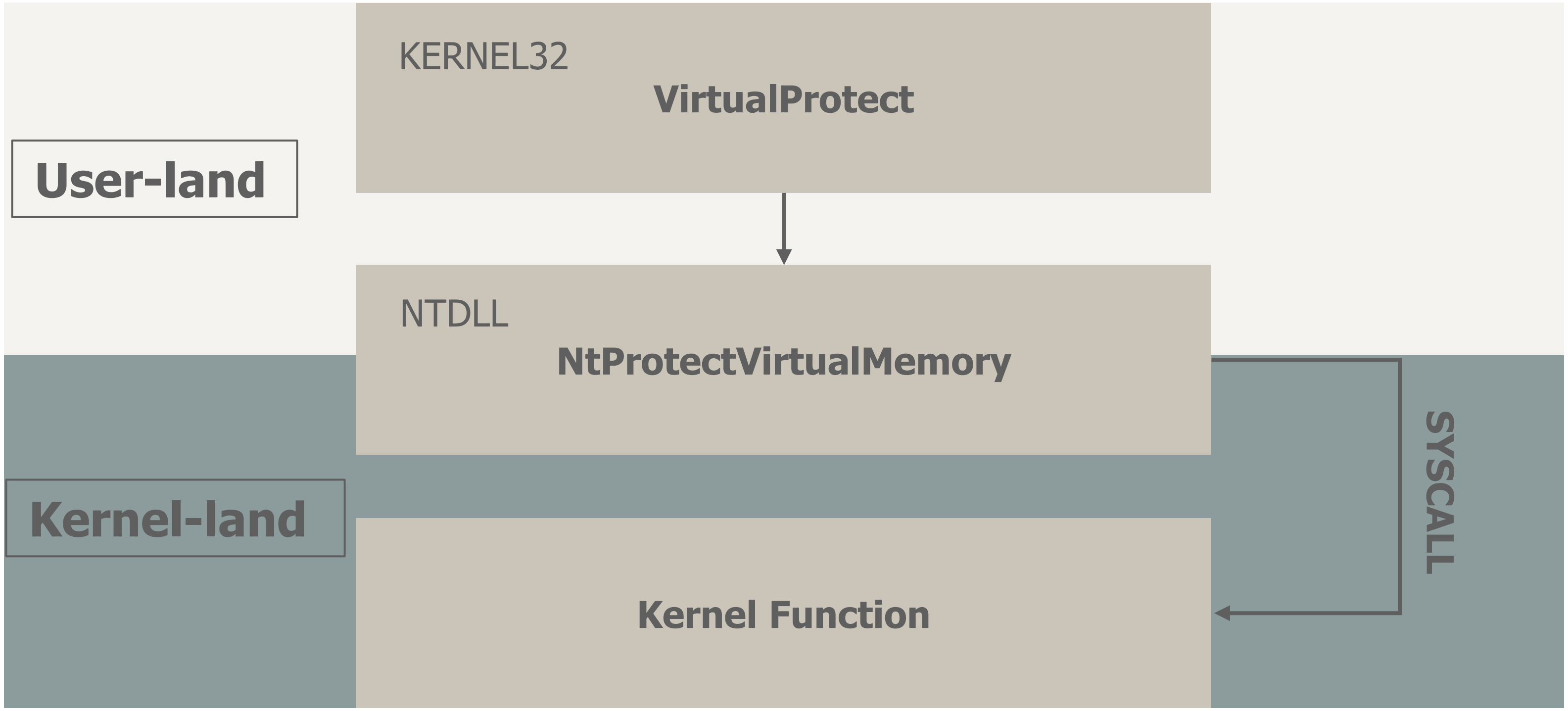
- Placing hooks on sensitive functions such as `CreateRemoteThread` or `NtAllocateVirtualMemory` allows the EDR to prevent their execution



EDR hooks 101
> Hooks, Userland and KernelLand

Userland VS KernelLand

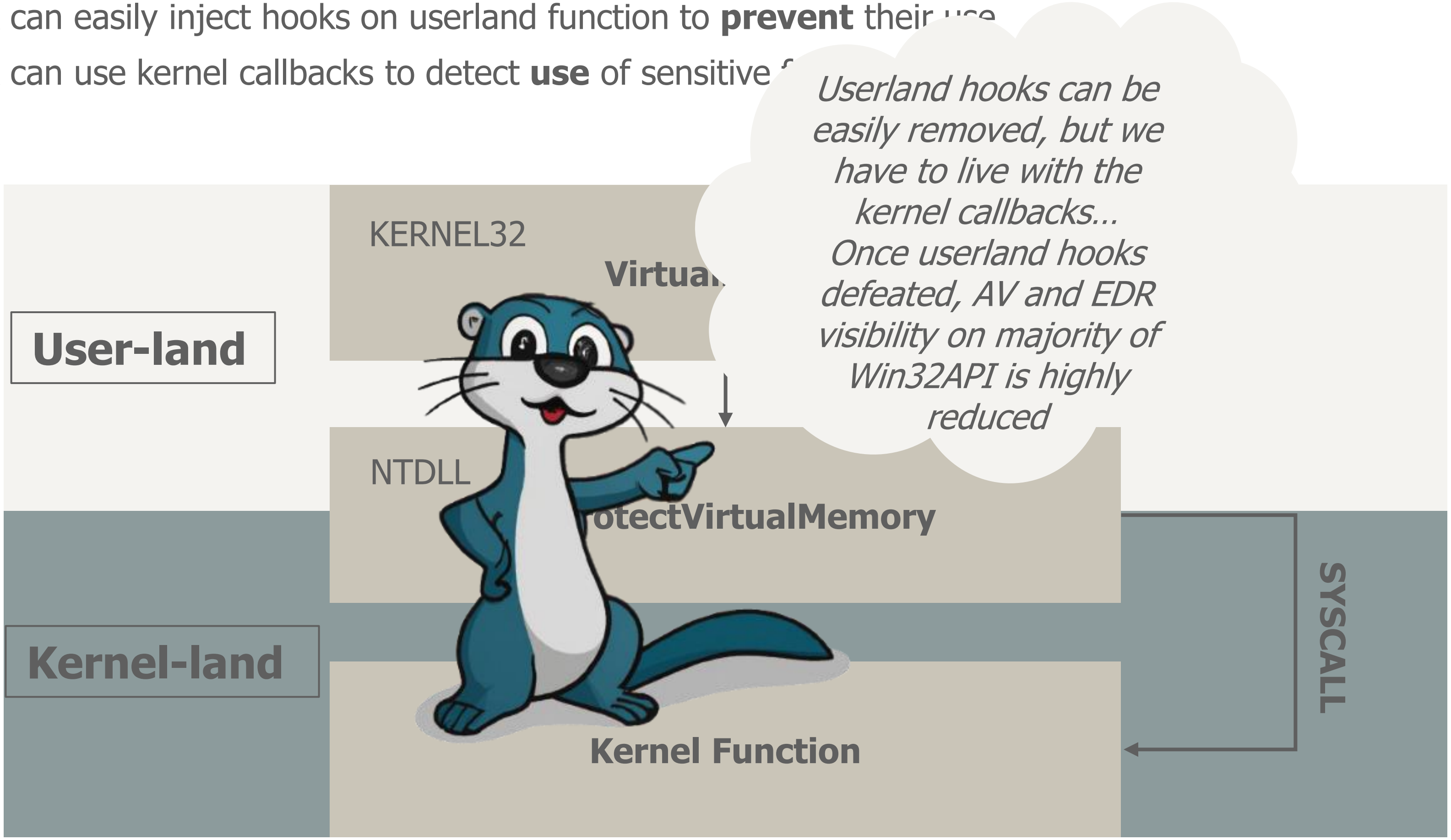
- EDR can easily inject hooks on userland function to **prevent** their use
- EDR can use kernel callbacks to detect **use** of sensitive functions



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 visibility on 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

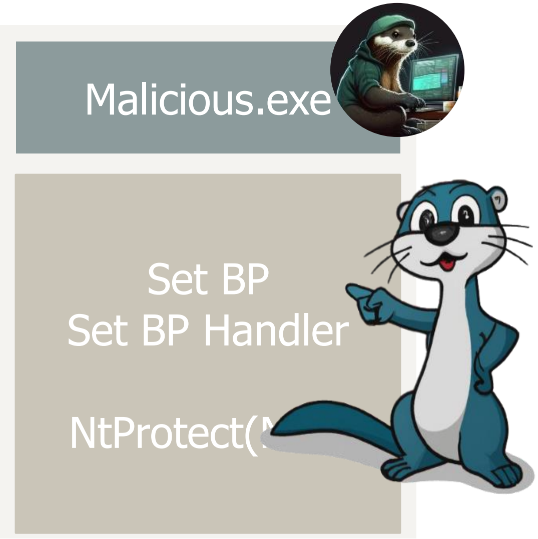
This is not a dehooking technique.
The EDR hook is neither modified nor deleted.

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

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

Bypass userland hooking
> Debugging



Malicious.exe

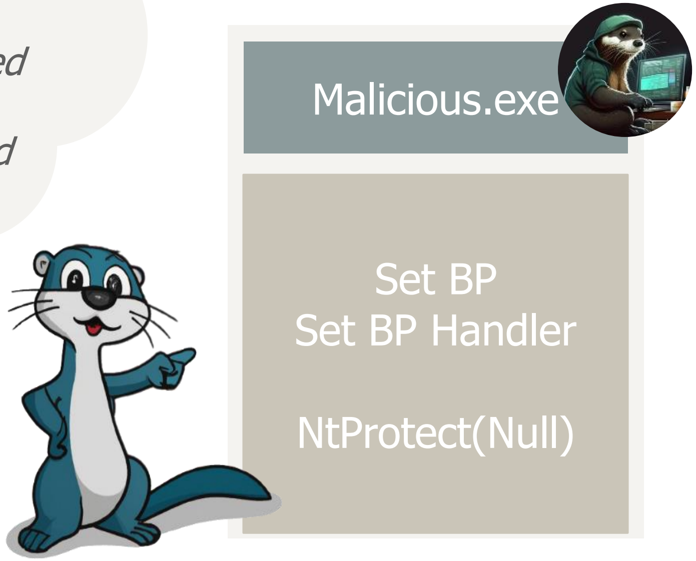
Set BP
Set BP Handler

NtProtect(

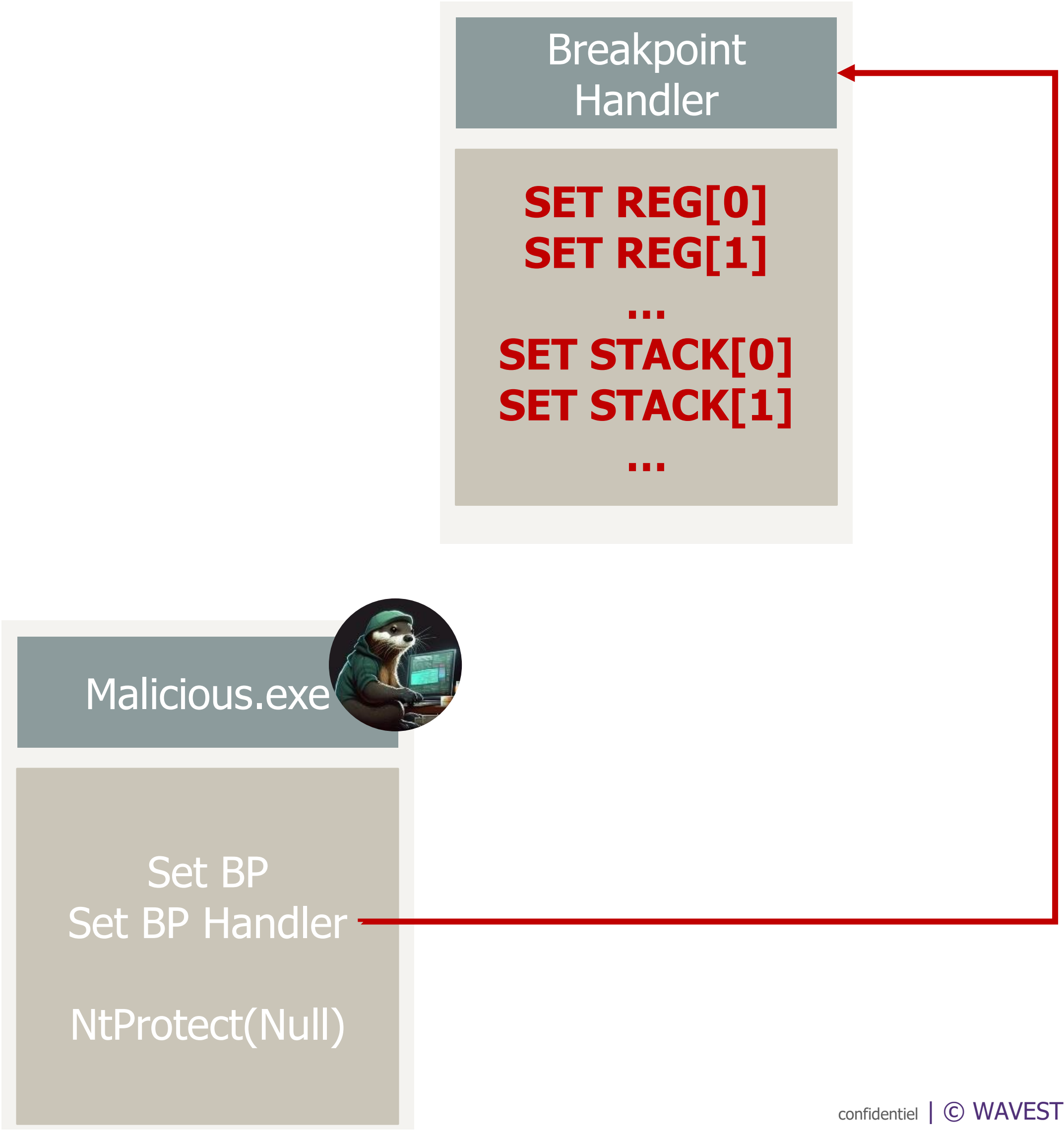
*A breakpoint is set to be triggered when the **SYSCALL** instruction is going to be executed. This is done by setting the **Dr0, Dr7 and Dr6** context registers*

Bypass userland hooking > Debugging

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



Bypass userland hooking
> Debugging



Breakpoint Handler

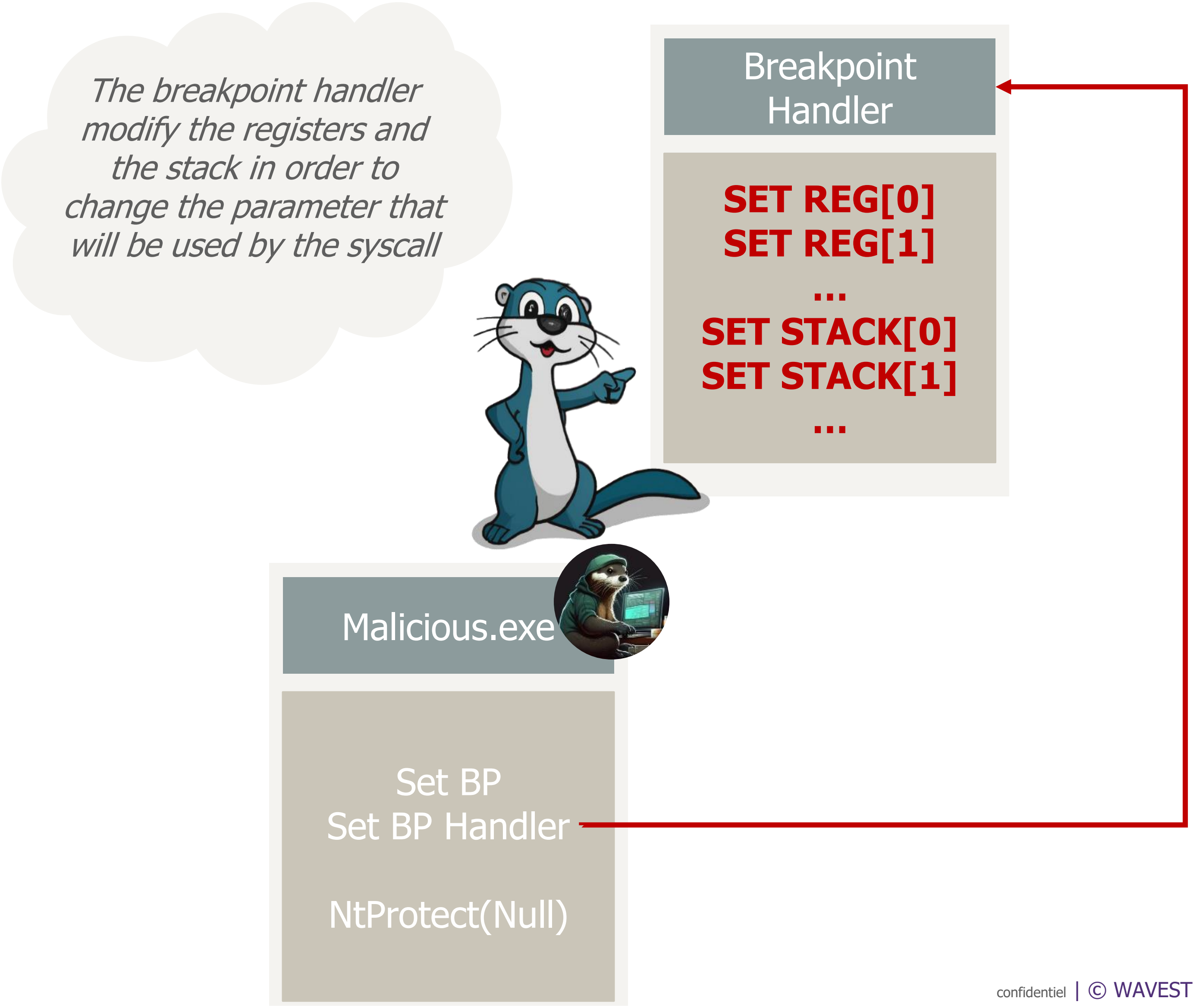
SET REG[0]
SET REG[1]
…
SET STACK[0]
SET STACK[1]
…

Malicious.exe

Set BP
Set BP Handler

NtProtect(Null)

Bypass userland hooking
> Debugging



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

Breakpoint Handler

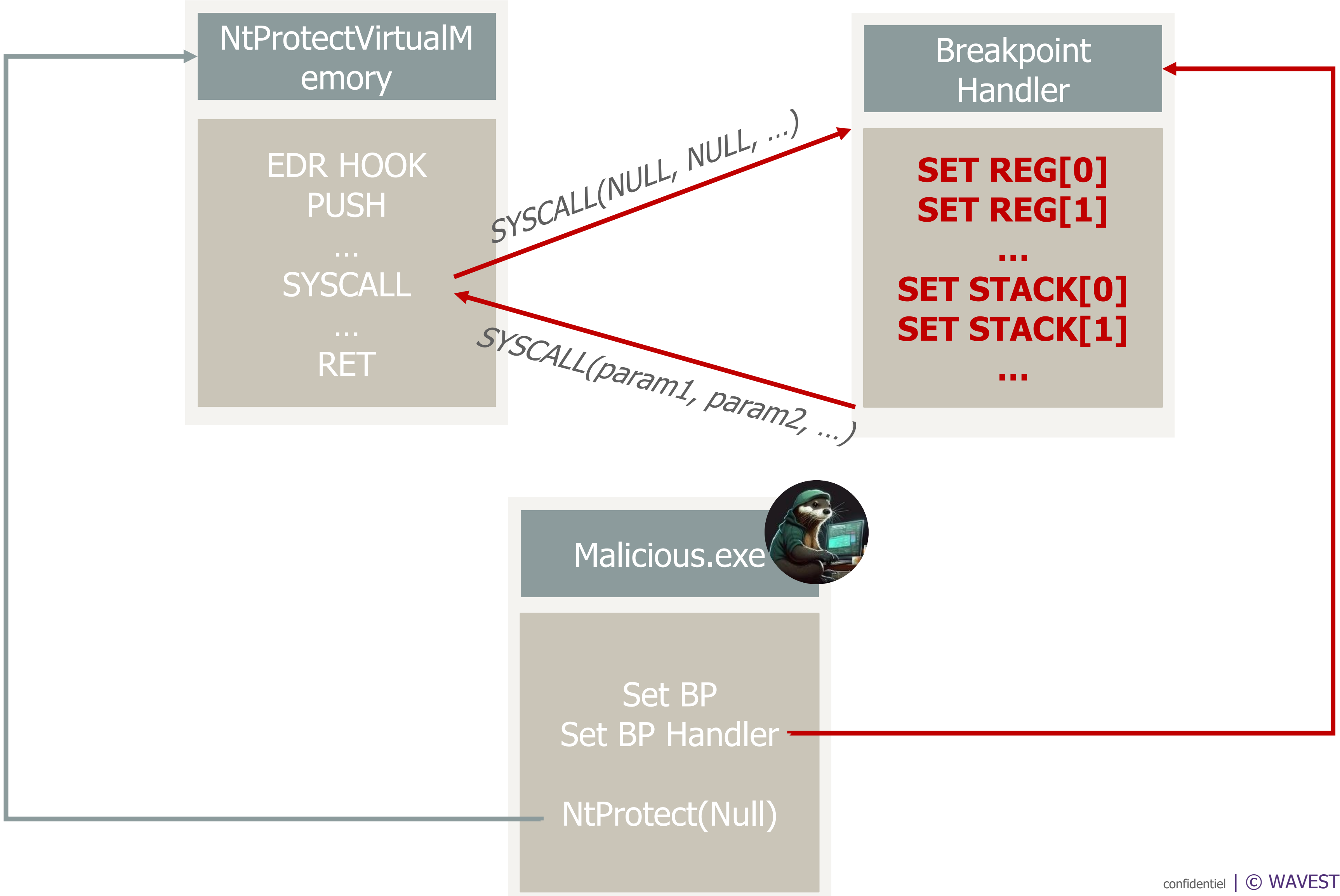
SET REG[0]
SET REG[1]
…
SET STACK[0]
SET STACK[1]
…

Malicious.exe

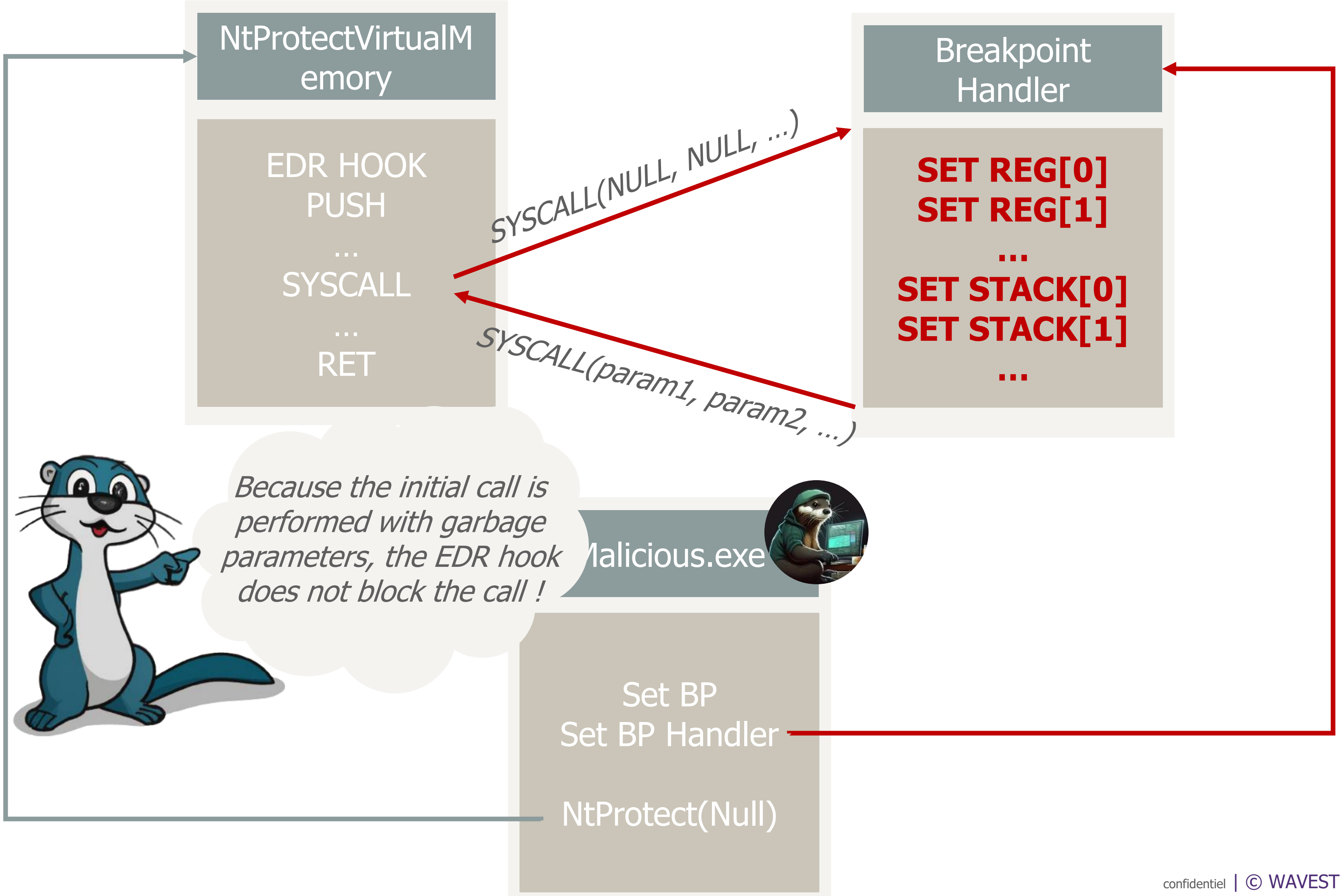
Set BP
Set BP Handler

NtProtect(Null)

Bypass userland hooking > Debugging



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)

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