ENSILO

SILO YOUR DATA FROM THREAT ACTORS

www.ensilo.com

Who?

- Udi Yavo
  - CTO and Co-Founder, enSilo
  - Former CTO, Rafael Cyber Security Division
  - Low Level Researcher
  - Author on BreakingMalware

- Tomer Bitton
  - VP Research and Co-Founder, enSilo
  - Low Level Researcher, Rafael Advanced Defense Systems
  - Malware Researcher
  - Author on BreakingMalware

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Agenda

- User-Mode Injections
  - Classic Code Injection Methods
  - Advanced Injections - PowerLoader
  - PowerLoaderEx
  - PowerLoaderEx64
- Kernel-Mode Injections
  - Common Injection Methods
  - Introducing Trap Frame Injection
  - Codeless Code Injection
  - Demo
- Summary

User-Mode Code Injections

Classic Code Injection Methods

- Vanila Injection - OpenProcess/ VirtualAllocEx/CreateRemoteThread/LoadLibrary
- GetThreadContext / SetThreadContext
- SetWindowsHookEx
- AppInit_DLLs and APPCERTDLLs
- ShimEng (InjectDll Tag)
- SetWindowLong
- QueueUserApc
- Replace Dependency DLL
- …

PowerLoader – POP/POP/INJECT

- Loader used in many different dropper families (Gapz / Redyms / Carberp / Vabushky ...)
- Technique was leaked with Carberp source code leak.
- First injection technique via Return Oriented Programming technique (ROP).
- “explorer.exe” is injected using Shell_TrayWnd / NtQueueApcThread (32bit / 64bit)
- Successfully bypassed most HIPSs solutions

1. http://www.welivesecurity.com/2013/03/19/gapz-and-redyms-droppers-based-on-power-loader-code/
2. http://www.slideshare.net/matrosov/advanced-evasion-techniques-by-win32gapz
3. https://www.virusbtn.com/virusbulletin/archive/2012/10/vb201210-code-injection
4. http://www.malwaretech.com/2013/08/powerloader-injection-something-truly.html

➢ Samples and valuable info - http://www.kernelmode.info/forum/

PowerLoader – Abusing Shared Sections

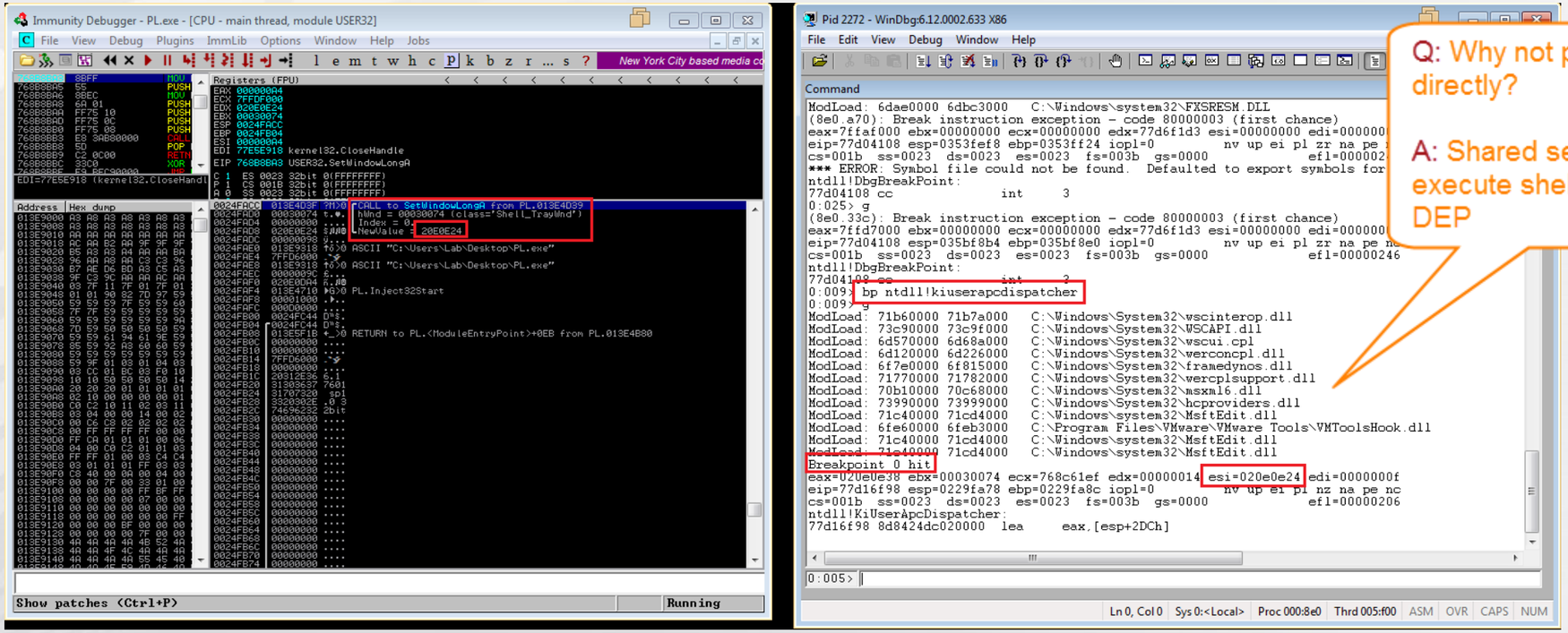
- Explorer.exe has several shared memory sections it uses:

```
.rdata:00407430 aBasenamedobjec:                        ; DATA XREF: sub_404790+C↑o
.rdata:00407430                 unicode 0, <\BaseNamedObjects\ShimSharedMemory>,0
.rdata:00407476                 align 4
.rdata:00407478 aBasenamedobj_0:                        ; DATA XREF: sub_404790+1A↑o
.rdata:00407478                 unicode 0, <\BaseNamedObjects\windows_shell_global_counters>,0
.rdata:004074D8 aBasenamedobj_1:                        ; DATA XREF: sub_404790+21↑o
.rdata:004074D8                 unicode 0, <\BaseNamedObjects\MSCTF.Shared.SFM.MIH>,0
.rdata:00407526                 align 4
.rdata:00407528 aBasenamedobj_2:                        ; DATA XREF: sub_404790+28↑o
.rdata:00407528                 unicode 0, <\BaseNamedObjects\MSCTF.Shared.SFM.AMF>,0
.rdata:00407576                 align 4
.rdata:00407578 aBasenamedobj_3:                        ; DATA XREF: sub_404790+2F↑o
.rdata:00407578                 unicode 0, <\BaseNamedObjects\UrlZonesSM_Administrator>,0
.rdata:004075CE                 align 10h
.rdata:004075D0 aBasenamedobj_4:                        ; DATA XREF: sub_404790+36↑o
.rdata:004075D0                 unicode 0, <\BaseNamedObjects\UrlZonesSM_SYSTEM>,0
```

- By calling NtOpenSection and ZwMapViewOfSection PowerLoader maps one of the shared sections to its own address space
- PowerLoader finds the address of the shared section in explorer by calling VirtualQueryEx / ReadProcessMemory / RtlCompareMemory combination
- The shellcode is than written to the shared section. There is no need for VirtualAllocEx / WriteProcessMemory which is monitored by many HIPS

PowerLoader - Triggering Code Execution

- One of the Windows in Explorer.exe is Shell_TrayWnd
- The Shell_TrayWnd window has a pointer to CTray class object which handles messages to the window
- Once the Shell_TrayWnd window is found PowerLoader uses SetWindowLongPtr to replace the CTray object with a malicious CTray Object in the shared section
- The shared section is not executable thus ROP is needed
- The virtual table of the malicious CTray object point to KiUserApcDispatcher routine which eventually trigger the ROP execution
- To trigger code execution SendNotifyMessage is called with WM_PAINT message



ENSILO
SILO YOUR DATA FROM THREAT ACTORS

PowerLoader - ROP n’ Roll

- PowerLoader uses a complex ROP chain
- Basic steps by the ROP chain (High-Level):
  - Direction Flag Set
  - Copy ROP chain to stack
  - Direction Flag Clear
  - Use WriteProcessMemory to overwrite ntdll!atan function (this function is probably not used in the context of explorer.exe)
  - Pass control to the overwritten ntdll!atan function

PowerLoader - Quick Summary

- Advanced Injection technique
- Completely bypassed most HIPS at the time
- Using Exploit-Like techniques:
  - The first Injection technique to use ROP (to our knowledge)
  - Overwrites atan function using WriteProcessMemory
  - No similar 64-bit version (to our knowledge)
  - Try to run vs EMET and see what happens…
- Very application specific:
  - Targets Shell_TrayWnd of explorer.exe
  - Uses specific shared sections
- Doesn’t use direct code injections

Introducing PowerLoaderEx

Goals:

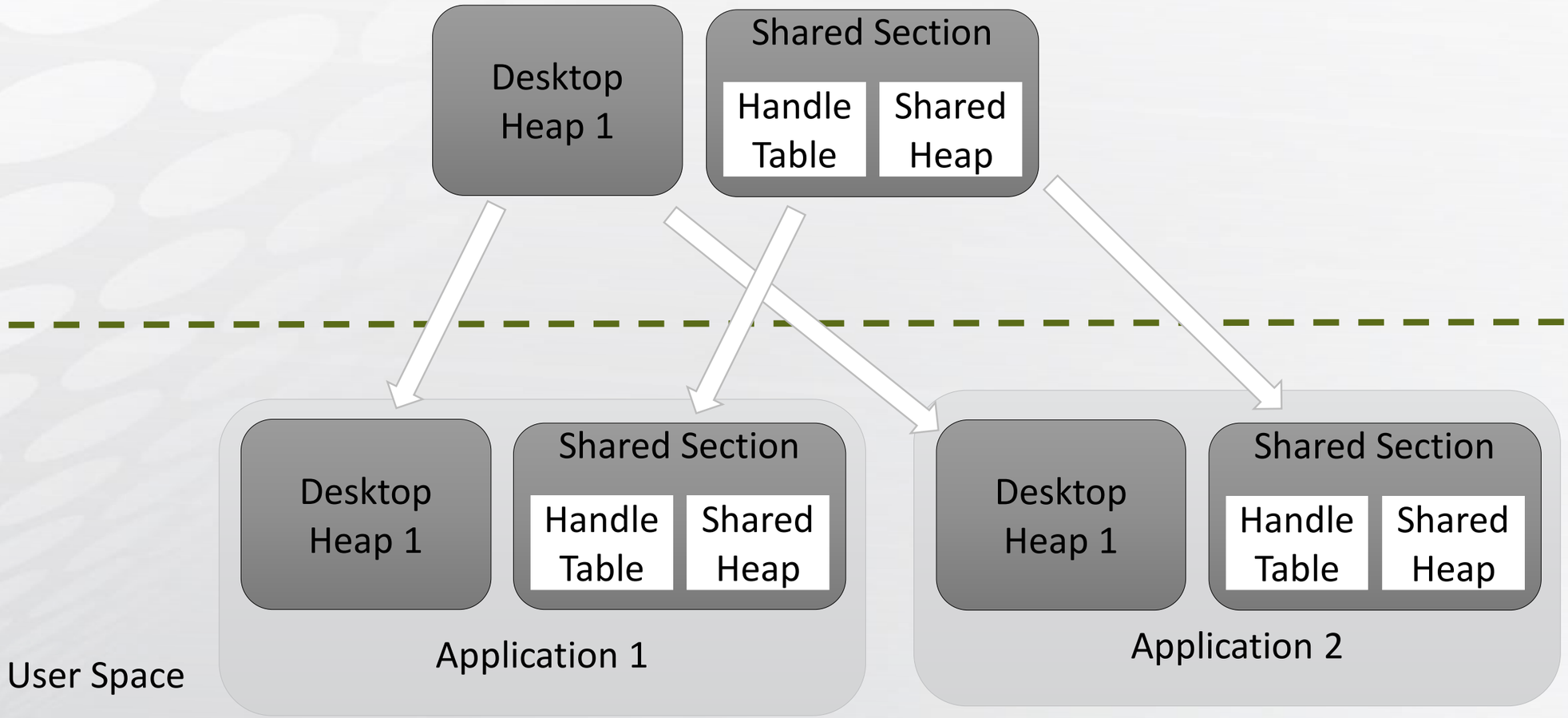
- Remove dependency in Explorer.exe shared sections (more generic)
- Make a 64-bit version
- Bonus: Do it without reading memory from the target process

UI Shared Memory - Reminder

- UI Objects are stored in one of three places:
  - Session Pool – Allocated per user logon
  - Desktop Heap – Stores objects specific to a given Desktop
  - Shared Heap – Handle Table and object relevant to all Desktops
- The heaps are visible to user-mode via shared-memory
- Commonly used for exploiting privilege escalation vulnerabilities
- Extensively documented in:
  “Windows Hooks Of Death: Kernel Attacks through User-Mode Callbacks” – By Tareji Mandt

UI Shared Memory

Kernel Space



User Space

PowerLoaderEx – Desktop Heap as Shared Section

- Desktop Heaps are better as shared section because they are mapped to any process on a given desktop
- Writing arbitrary data to the Desktop Heap is easy:
  - Create a window with enough extra bytes
  - Use SetWindowLongPtr to write the payload
- But how do know where it resides in a target process?

Finding Target Process's Desktop Heap

- Find the Desktop Heap in our process and find its region size
- Enumerate explorer.exe memory regions by calling VirtualQuery.
- The target's Desktop Heap is found if the region characteristics are:
  - State: MEM_COMMIT
  - Type: MEM_MAPPED
  - Protect: PAGE_READ_ONLY
  - RegionSize is equal to the previously obtained heapSize.

Double Check – Avoiding False Positives

- We had **NO** false positives in finding the shared desktop heap (local process and target process)
- The following check will remove any false-positive:

```
while (VirtualQueryEx(ProecssHandle, Addr, &MemInfo, sizeof(MemInfo)))
{
    if (MemInfo.Protect = PAGE_READONLY && MemInfo.Type == MEM_MAPPED && MemInfo.State == MEM_COMMIT && MemInfo.RegionSize == HeapSize)
    {
        // Double check.
        if (!VirtualProtectEx(ProecssHandle, Addr, 0x1000, PAGE_READWRITE, &OldProt))
        {
            // return section information.
            return MemInfo.BaseAddress;
        }
        else
        {
            VirtualProtectEx(ProecssHandle, Addr, 0x1000, OldProt, &OldProt);
        }
    }
    Addr += MemInfo.RegionSize;
}
```

The call will fail if it's the desktop heap and otherwise will succeed.

* Note: using the double check version will require calling NtOpenProcess with QUERY_INFORMATION and PROCESS_VM_OPERATION

Finding The Gadgets

- We do the gadget lookup in our own process
- Load and scan only modules that we know that are always loaded in the target process (i.e. explorer.exe)
- Use only modules that are very likely to have the same base in both processes:
  - ntdll.dll
  - Kernel32.dll
  - Kernelbase.dll
  - User32.dll
  - Shell32.dll

PowerLoaderEx - Look Ma, No Read!

1. Create a window and find it in the shared desktop heap
2. Find the target process (Explorer.exe) and find where the desktop heap is mapped, requires only PROCESS_QUERY_INFORMATION privileges
3. Scan for the required gadgets
4. Write the payload to the shared desktop heap using SetWindowLongPtr
5. Continue like normal PowerLoader

* Note: If a different target process has more than one Desktop Heap mapped read will be needed

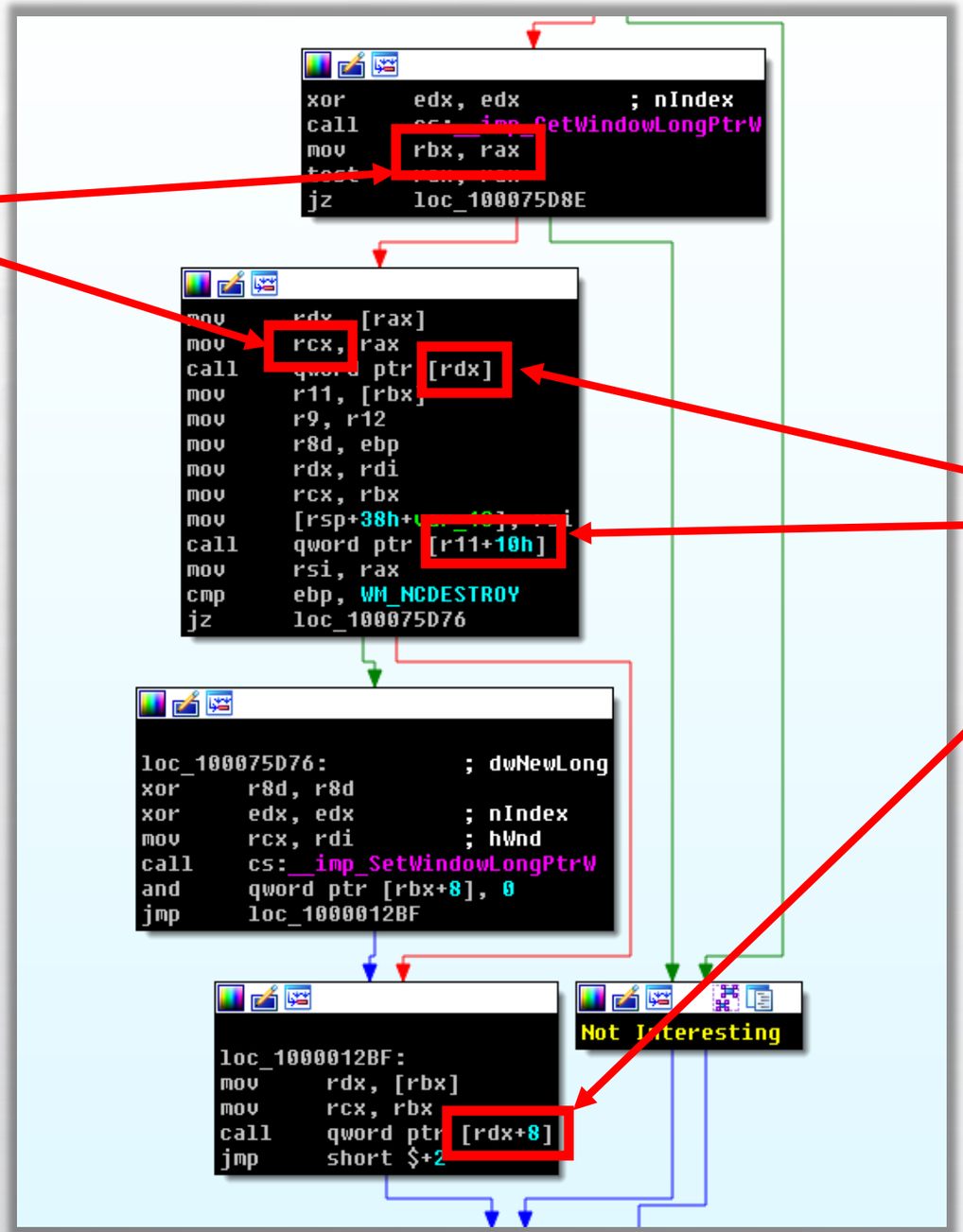
PowerLoaderEx64 - What About 64-bit?

- Challenges:
  - Much harder to find useful gadgets
  - No code writing allowed
  - No reads allowed

PowerLoaderEx64 – Remember the CTray Object?

CImpWndProc::s_WndProc

The Controlled
Ctray Object



Virtual Calls

PowerLoaderEx64 - Strategy

- Find some function that can be manipulated to call LoadLibrary with controlled first argument
- Make sure no crashes occur after/during library loading
- To make things simple point 2 of the 3 virtual calls to “Do Nothing” functions (Ret opcode)

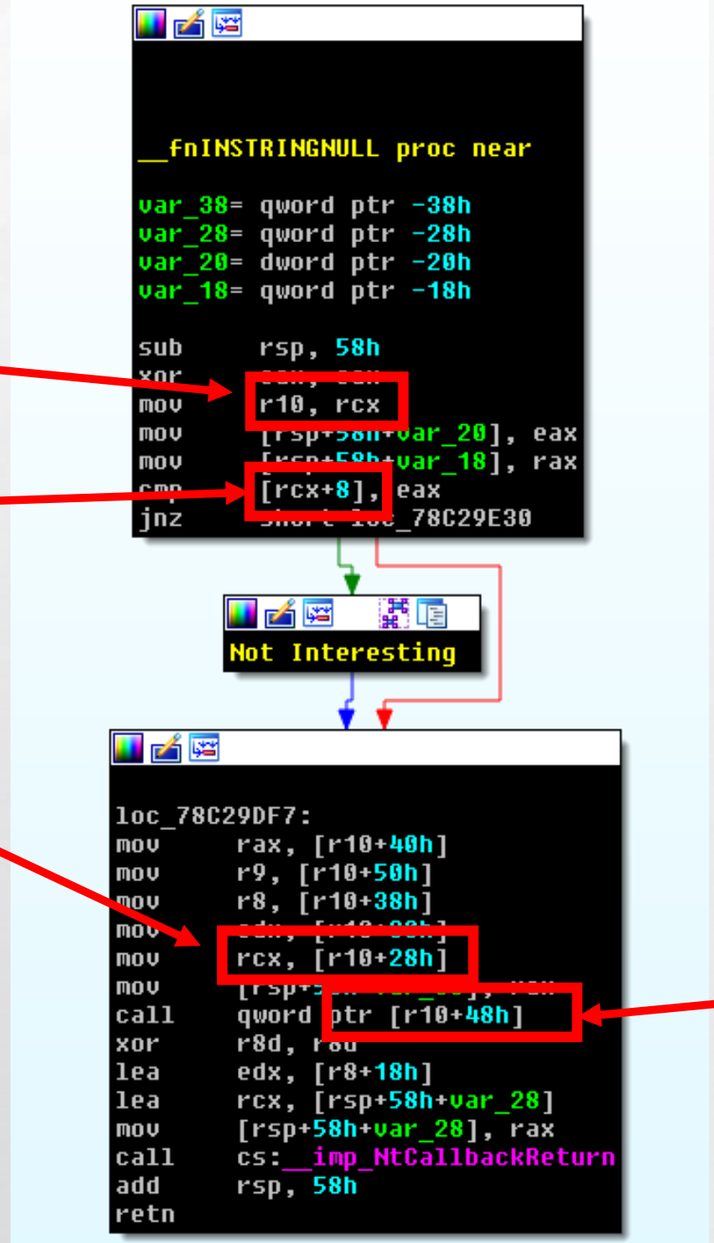
PowerLoaderEx64 – User32 Callbacks To Rescue

The Controlled Ctray Object

Should be NULL

Pointer to DLL Path

Pointer to LoadLibraryA



PowerLoaderEx64 - Recap

1. Create a window and find it in the shared desktop heap
2. Find the target process (Explorer.exe) and find where the desktop heap is mapped, requires only PROCESS_QUERY_INFORMATION privileges
3. Write the malicious CTray object to the shared desktop heap using SetWindowLongPtr
4. Replace the Shell_TrayWnd window's CTray object using SetWindowLongPtr
5. Use SendNotifyMessage to trigger LoadLibrary

PowerLoaderEx64 source code will be on BreakingMalware

http://BreakingMalware.com

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Kernel-To-User Code Injections

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Introduction - Kernel-To-User Code Injections

- Mainly used for:
  - Injecting DLLs
  - Sandbox escapes – After exploiting privilege escalation vulnerability
  - Injecting to protected processes
- Fewer techniques exist than user-mode
- Less documented than user-mode techniques
- Used by both Malware and Software/Security vendors

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Common Injection Methods – User APC

- The most common Kernel-To-User injection method
- Used by lots of malwares:
  - TDL
  - ZERO ACCESS
  - Sandbox escape shellcodes
  - …
- Also used by lots of security products:
  - AVG
  - Kaspersky Home Edition
  - Avecto
  - …
- Documented:
  - Blackout: What Really Happened
  - Much more in forums and leaked source codes

Common Injection Methods – User APC

Basic Steps (There are several variations):

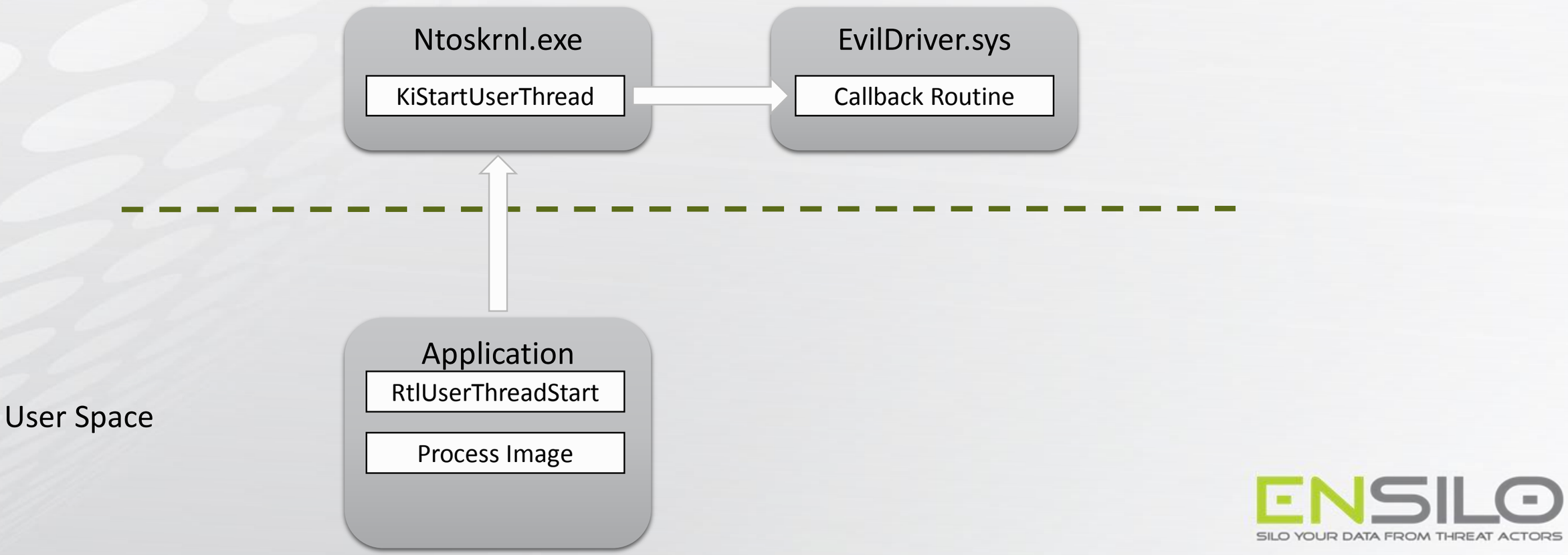
1. Register load image callback using PsSetLoadImageNotifyRoutine and wait for main module to load
2. Write payload that injects a dll using LdrLoadDll
   (Other variations use LoadLibrary)
3. Insert User APC using KeInsertQueueApc

Common Injection Methods – Entry Point Patching

- Not really common but worth mentioning
- Used by Duqu
- Fully documented in:
  http://binsec.gforge.inria.fr/pdf/Malware2013-Analysis-Diversion-Duqu-paper.pdf

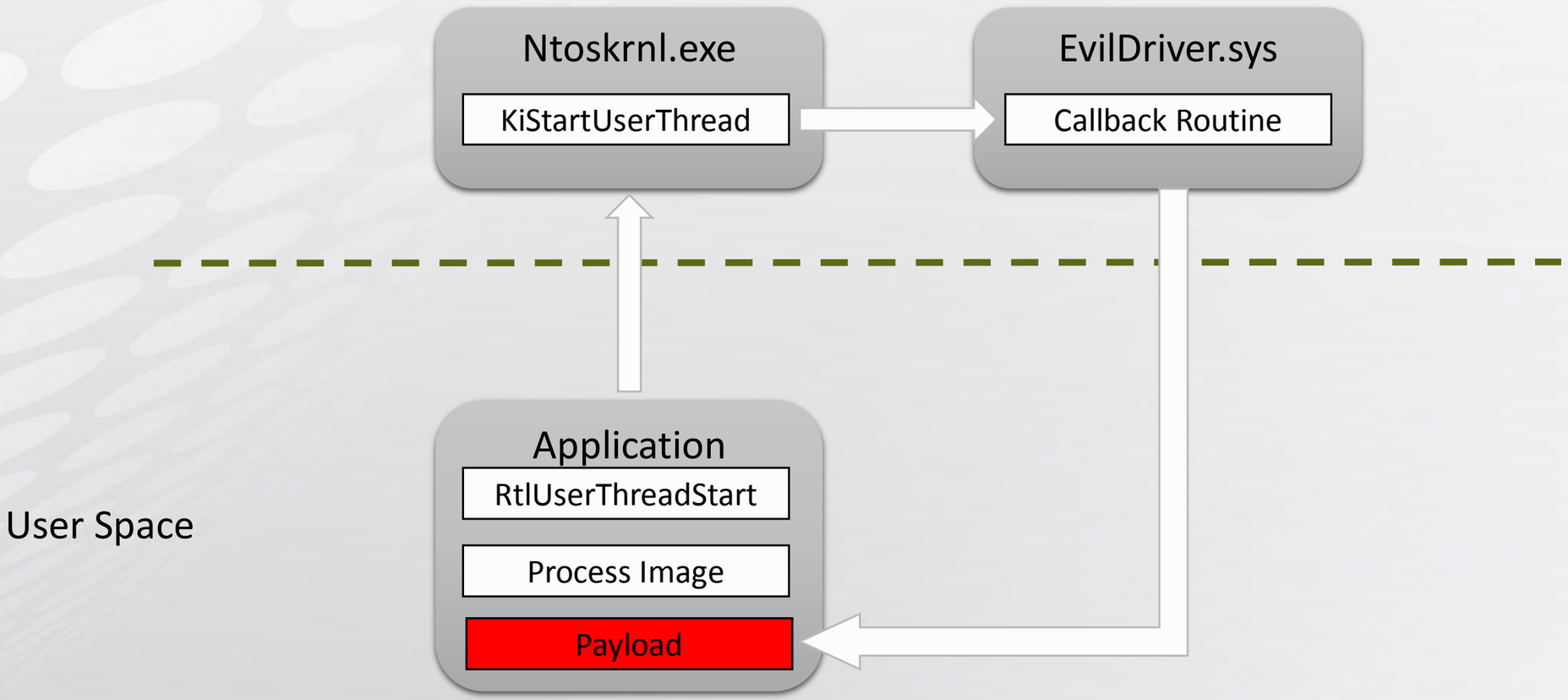
Common Injection Methods – Entry Point Patching

- Register load image callback using PsSetLoadImageNotifyRoutine and wait for main module to load



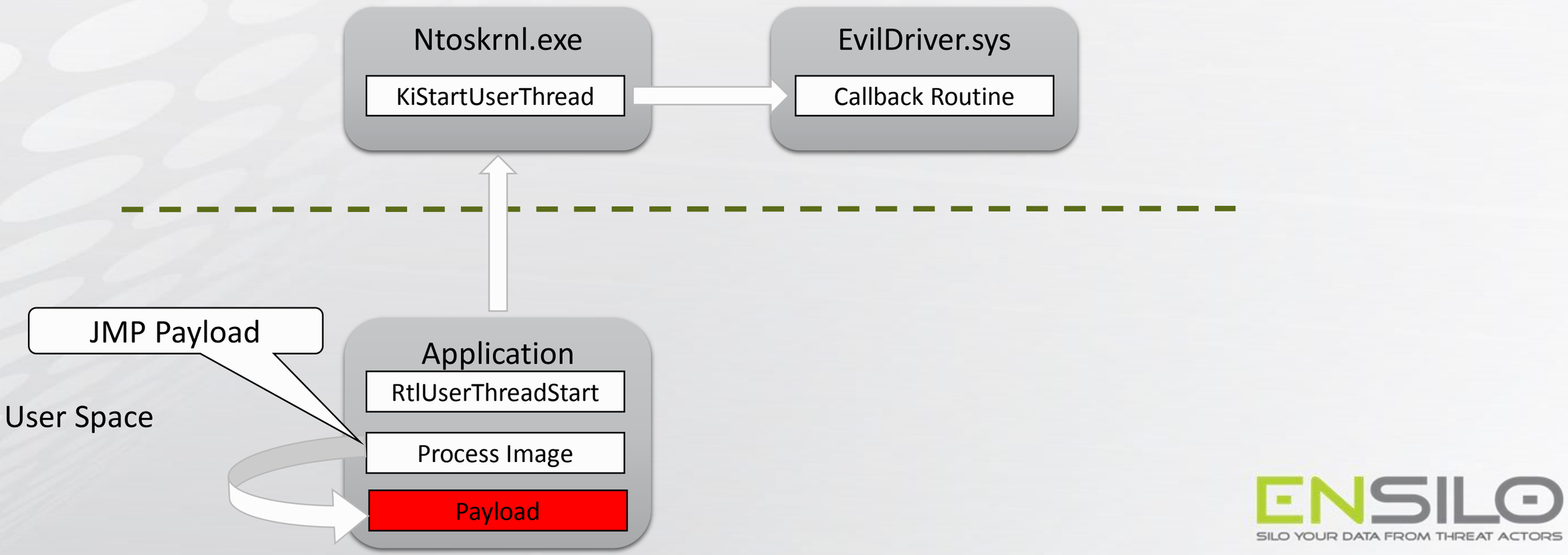
Common Injection Methods – Entry Point Patching

- Write the payload to the process address space



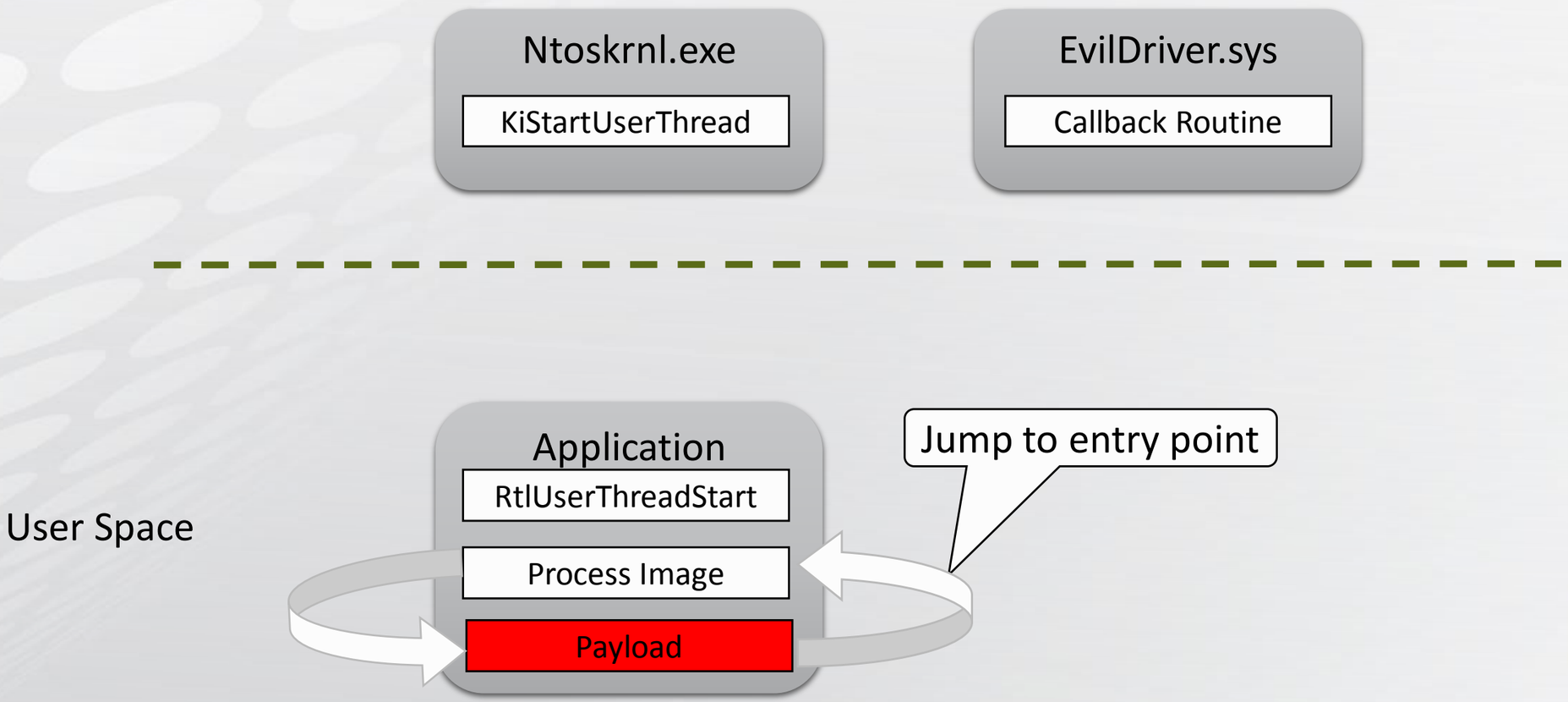
Common Injection Methods – Entry Point Patching

- Replace the image entry point with **JMP** to the new code



Common Injection Methods – Entry Point Patching

- The payload executes, fixes the entry point and jumps to it

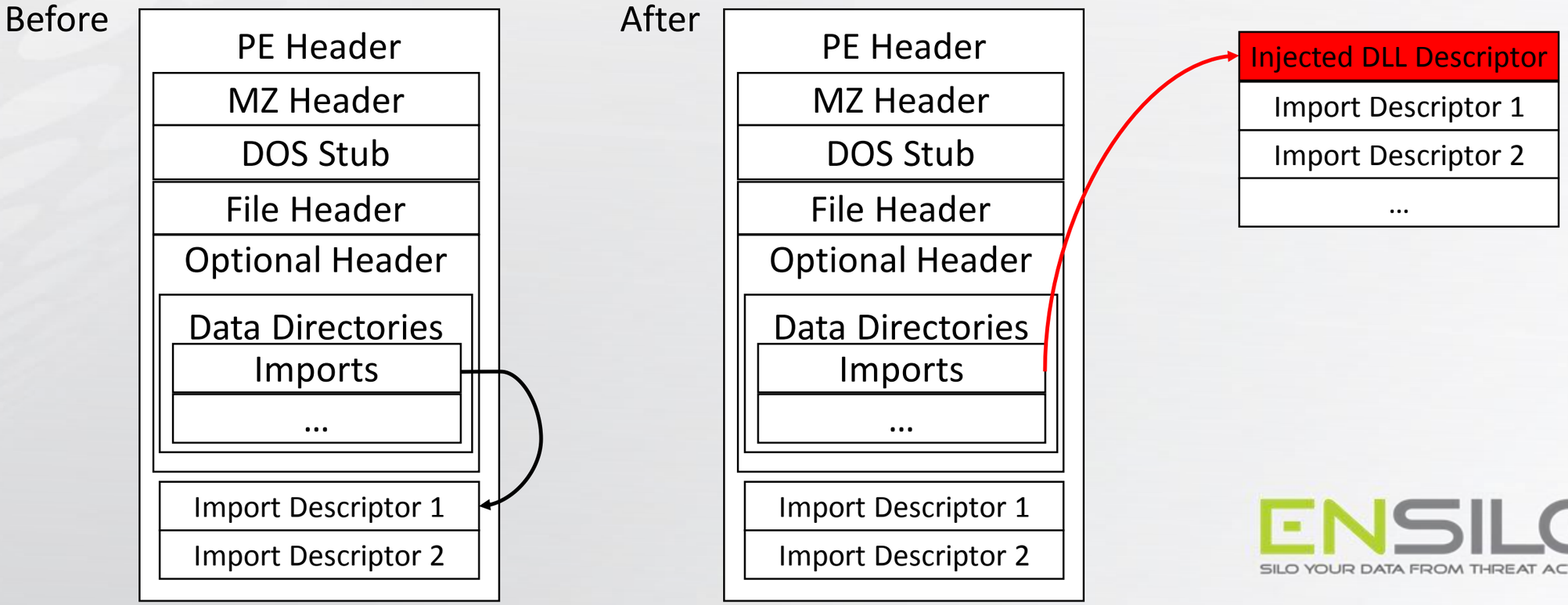


Common Injection Methods – Import Table Patching

- Undocumented (to our knowledge)
- Never been used by malware (to our knowledge)
- Used by software and security vendors:
  - Symantec
  - Trusteer
  - Microsoft App-V
- Similar method could probably use TLS data directory

Common Injection Methods – Import Table Patching

1. Register load image callback using PsSetLoadImageNotifyRoutine and wait for main module to load
2. Allocate memory for the new import table and copy old table with a new record for the injected DLL
3. Point the import data directory to the new table
4. When the DLL is loaded the original PE header is restored



ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Common Injection Methods – Quick Summary

- Kernel-To-User Injections are extensively used by both malware and security/software vendors
- Kernel injections are mainly used to inject a DLL to target(or all) processes
- All these methods require injection of some payload to user mode (except for Import Table Patching)
- All use PsSetLoadImageNotifyRoutine

Introducing Trap Frame Injection

- A new kernel-to-user injection technique
- Trap frames save the CPU user-mode state during exceptions or interrupt handling
- A Trap frame is created each time a system call is made and stored on the kernel stack
- The kernel structure used for trap frames is _KTRAP_FRAME
- The user-mode state is restored when the system call returns
- Current frame is stored in the TrapFrame field of _KTHREAD

KTRAP_FRAME

32-Bit

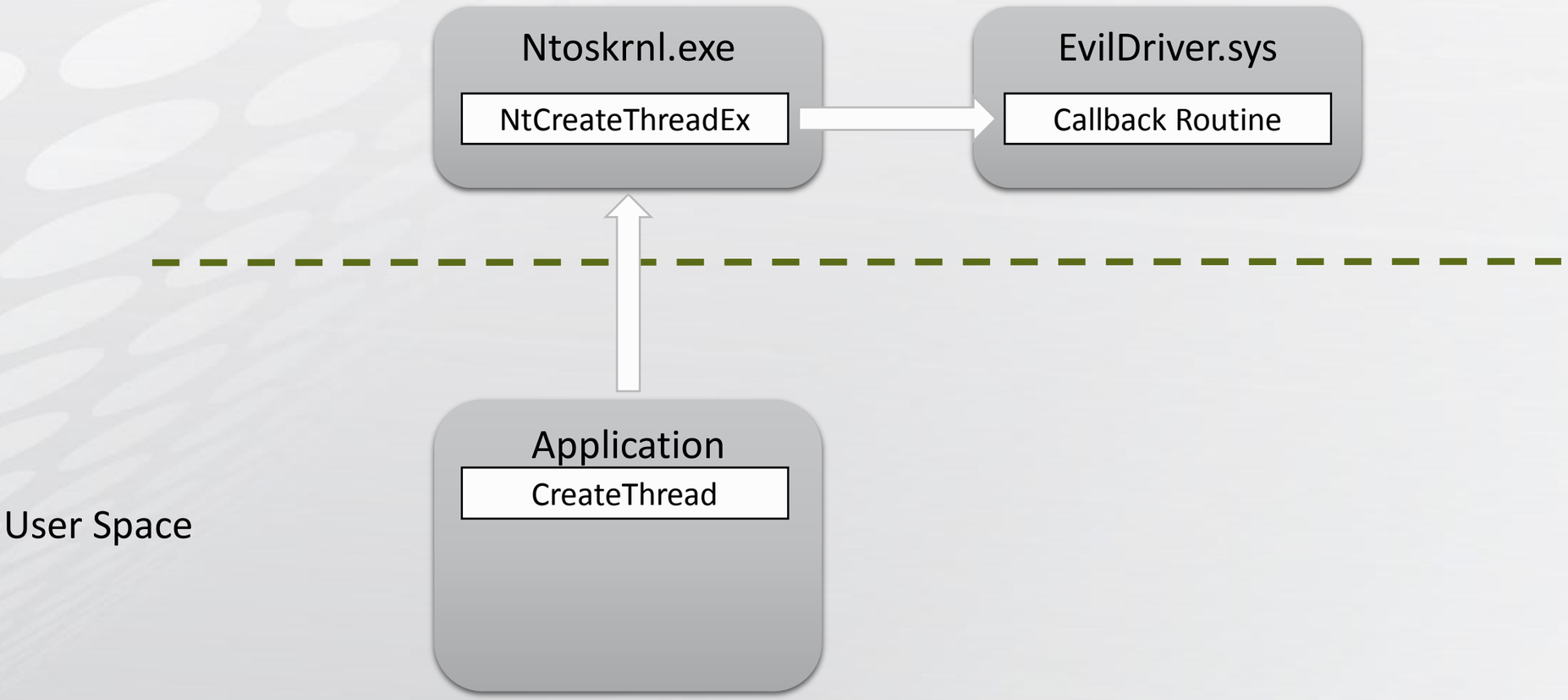
```
kd> dt _ktrap_frame
nt!_KTRAP_FRAME
   +0x000 DbgEbp           : Uint4B
   +0x004 DbgEip           : Uint4B
   +0x008 DbgArgMark       : Uint4B
   +0x00c DbgArgPointer    : Uint4B
   +0x010 TempSegCs        : Uint2B
   +0x012 Logging          : UChar
   +0x013 Reserved         : UChar
   +0x014 TempEsp          : Uint4B
   +0x018 Dr0              : Uint4B
   +0x01c Dr1              : Uint4B
   +0x020 Dr2              : Uint4B
   +0x024 Dr3              : Uint4B
   +0x028 Dr6              : Uint4B
   +0x02c Dr7              : Uint4B
   +0x030 SegGs            : Uint4B
   +0x034 SegEs            : Uint4B
   +0x038 SegDs            : Uint4B
   +0x03c Edx              : Uint4B
   +0x040 Ecx              : Uint4B
   +0x044 Eax              : Uint4B
   +0x048 PreviousPreviousMode : Uint4B
   +0x04c ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x050 SegFs            : Uint4B
   +0x054 Edi              : Uint4B
   +0x058 Esi              : Uint4B
   +0x05c Ebx              : Uint4B
   +0x060 Ebp              : Uint4B
   +0x064 ErrCode          : Uint4B
   +0x068 Eip              : Uint4B
   +0x06c SegCs            : Uint4B
   +0x070 EFlags           : Uint4B
   +0x074 HardwareEsp      : Uint4B
   +0x078 HardwareSegSs    : Uint4B
   +0x07c V86Es            : Uint4B
   +0x080 V86Ds            : Uint4B
   +0x084 V86Fs            : Uint4B
   +0x088 V86Gs            : Uint4B
```

64-Bit

```
kd> dt _ktrap_frame
nt!_KTRAP_FRAME
   ...
   +0x030 Rax              : Uint8B
   +0x038 Rcx              : Uint8B
   +0x040 Rdx              : Uint8B
   +0x048 R8               : Uint8B
   +0x050 R9               : Uint8B
   +0x058 R10              : Uint8B
   +0x060 R11              : Uint8B
   +0x068 GsBase           : Uint8B
   +0x068 GsSwap           : Uint8B
   ...
   +0x0d0 FaultAddress     : Uint8B
   +0x0d0 ContextRecord    : Uint8B
   +0x0d0 TimeStampCKCL    : Uint8B
   +0x0d8 Dr0              : Uint8B
   ...
   +0x130 SegDs            : Uint2B
   +0x132 SegEs            : Uint2B
   +0x134 SegFs            : Uint2B
   +0x136 SegGs            : Uint2B
   +0x138 TrapFrame        : Uint8B
   +0x140 Rbx              : Uint8B
   +0x148 Rdi              : Uint8B
   +0x150 Rsi              : Uint8B
   +0x158 Rbp              : Uint8B
   ...
   +0x168 Rip              : Uint8B
   +0x170 SegCs            : Uint2B
   ...
   +0x180 Rsp              : Uint8B
   ...
```

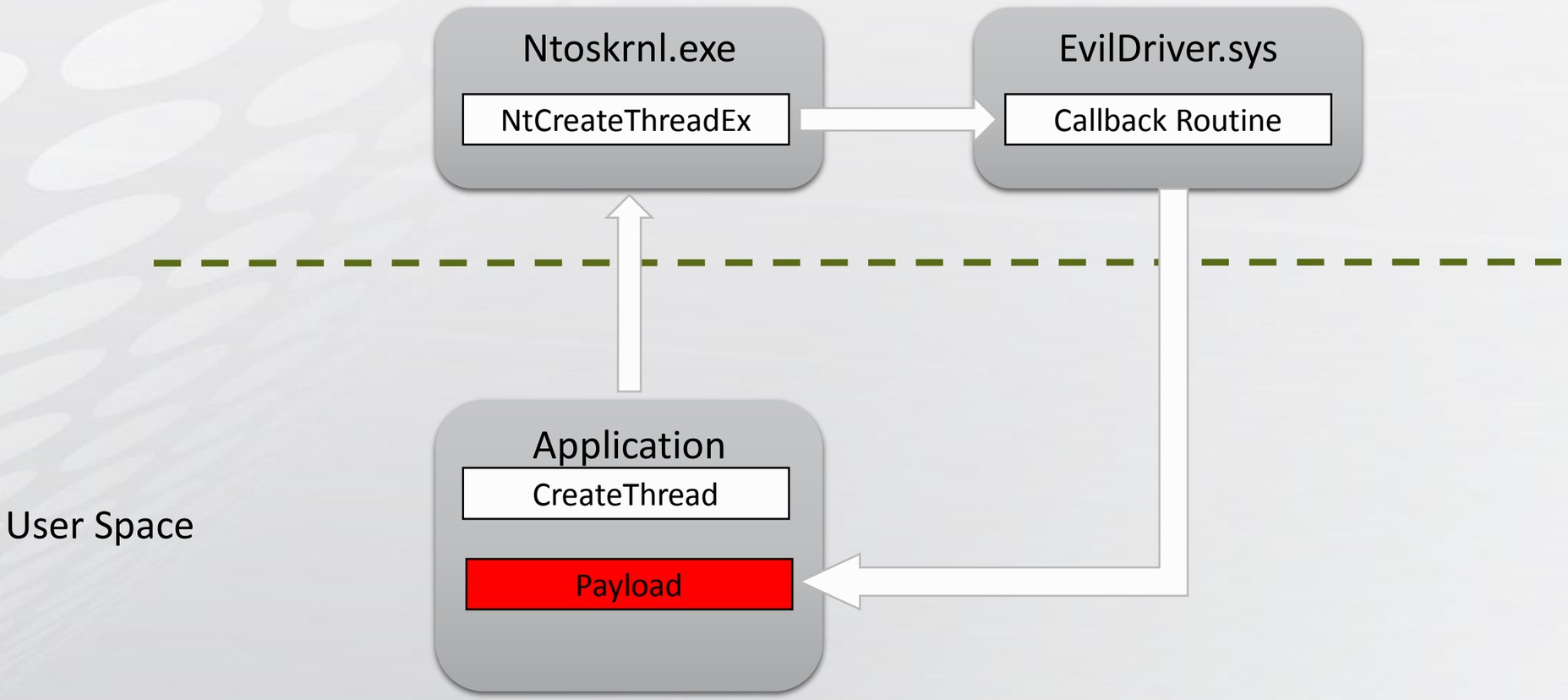
The Trivial Injection

- Wait for some callback that runs in context of a system call (Thread Creation, Registry Access,…)



The Trivial Injection

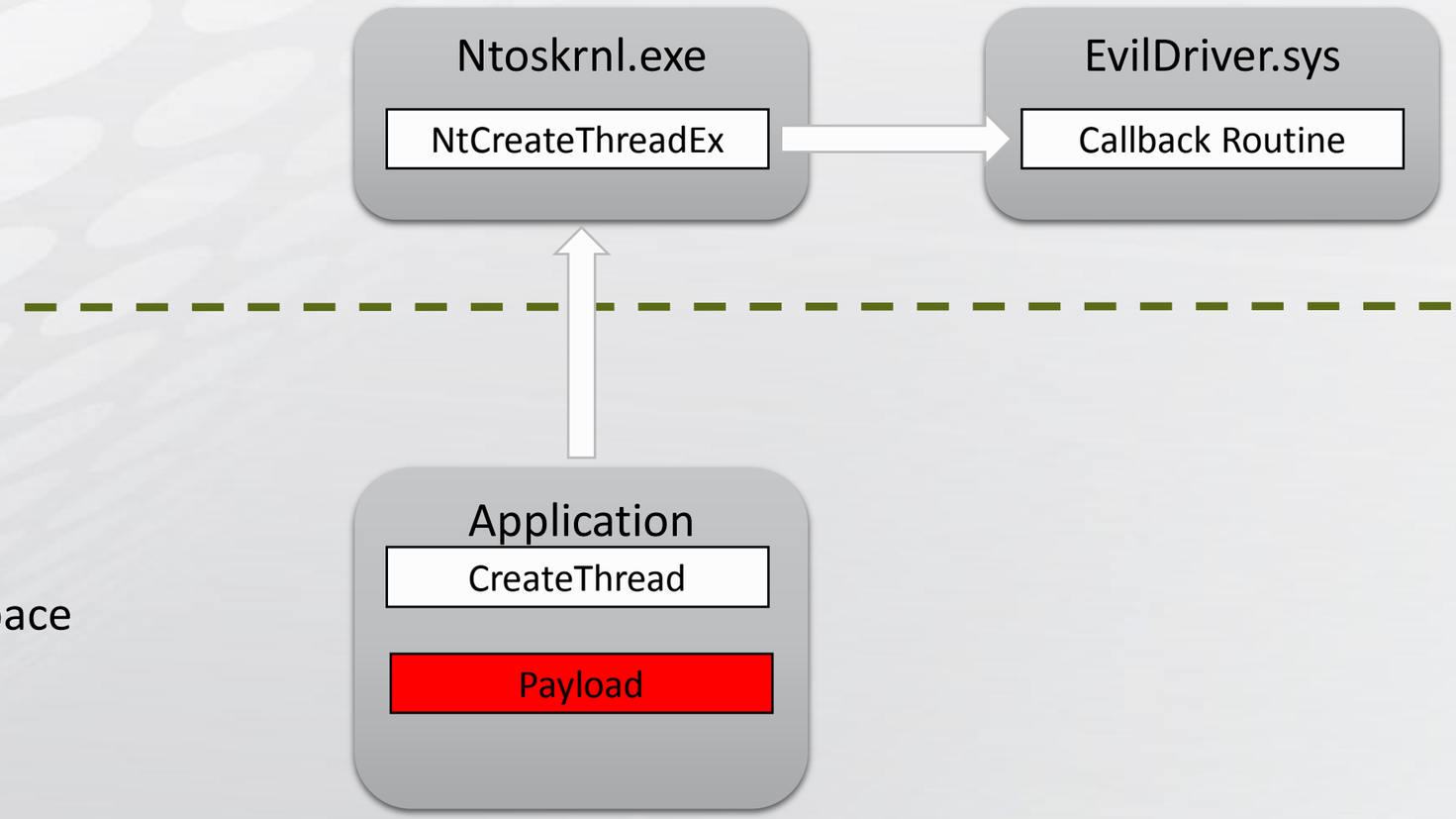
- Allocate payload in the target application



The Trivial Injection

- Fix the trap frame's saved RIP/EIP to the payload

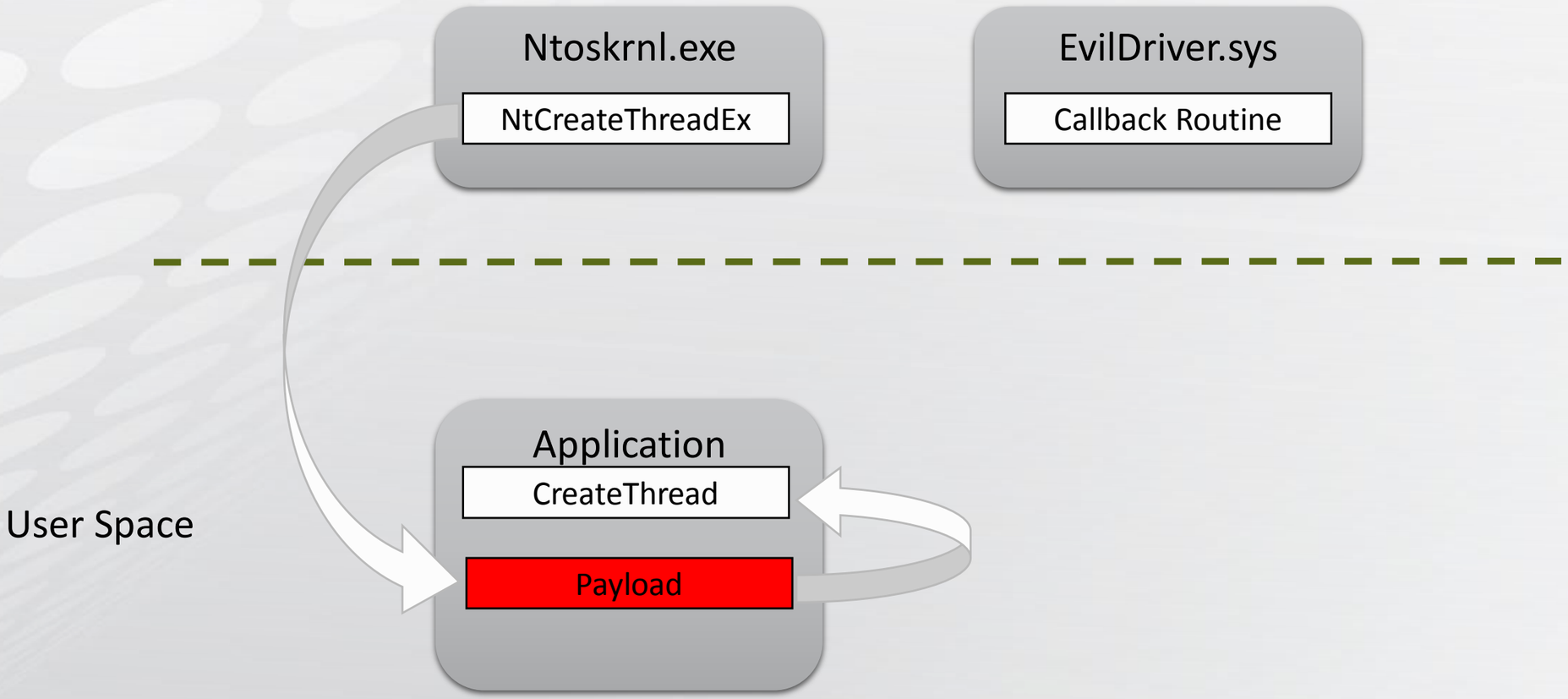
Kernel Space



User Space

The Trivial Injection

- The payload runs and restores normal execution



But We Promised a Codeless Code Injection…

Goals:

- Run code in the context of an arbitrary process:
  - Force IExplore.exe to send POST
  - Open remote shell
  - …
- Easy to write

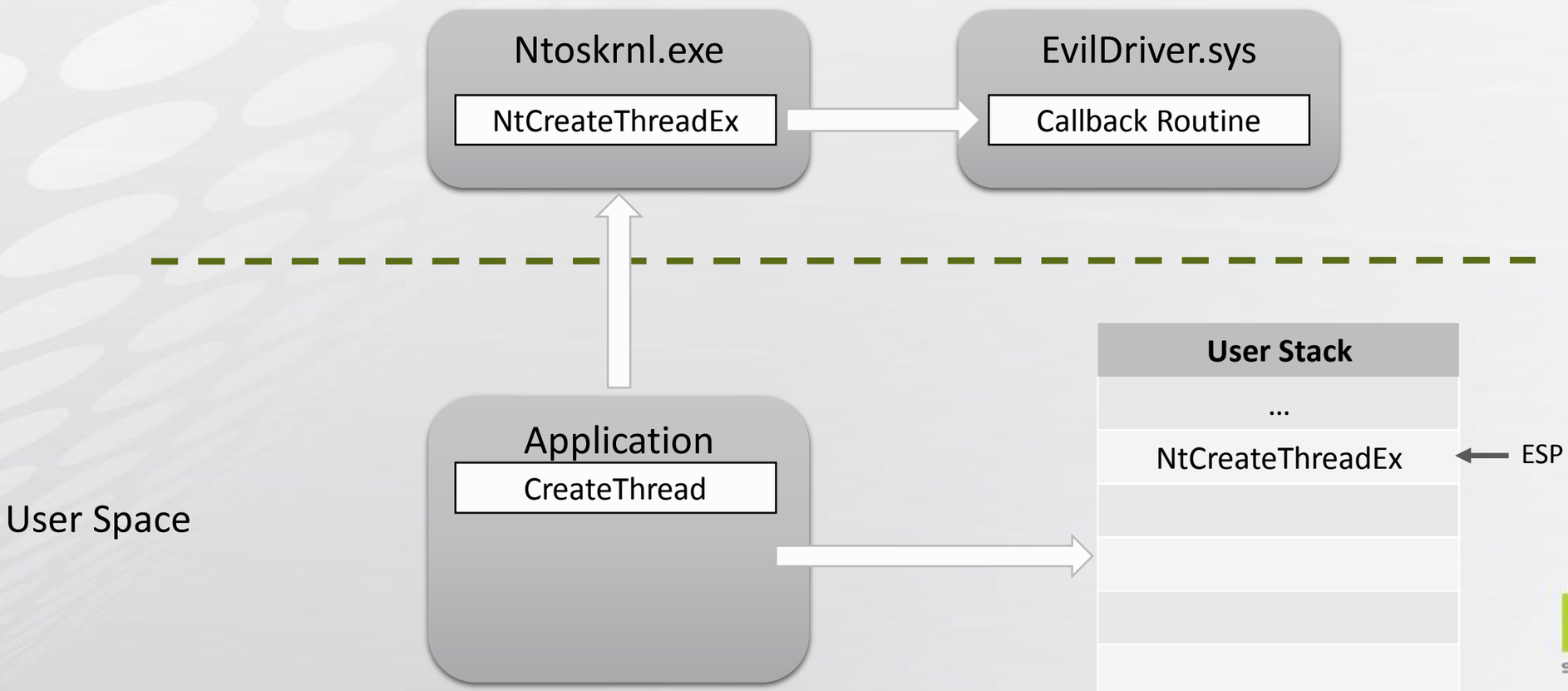
Limitation:

- No code injections to the process

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Codeless Code Injection

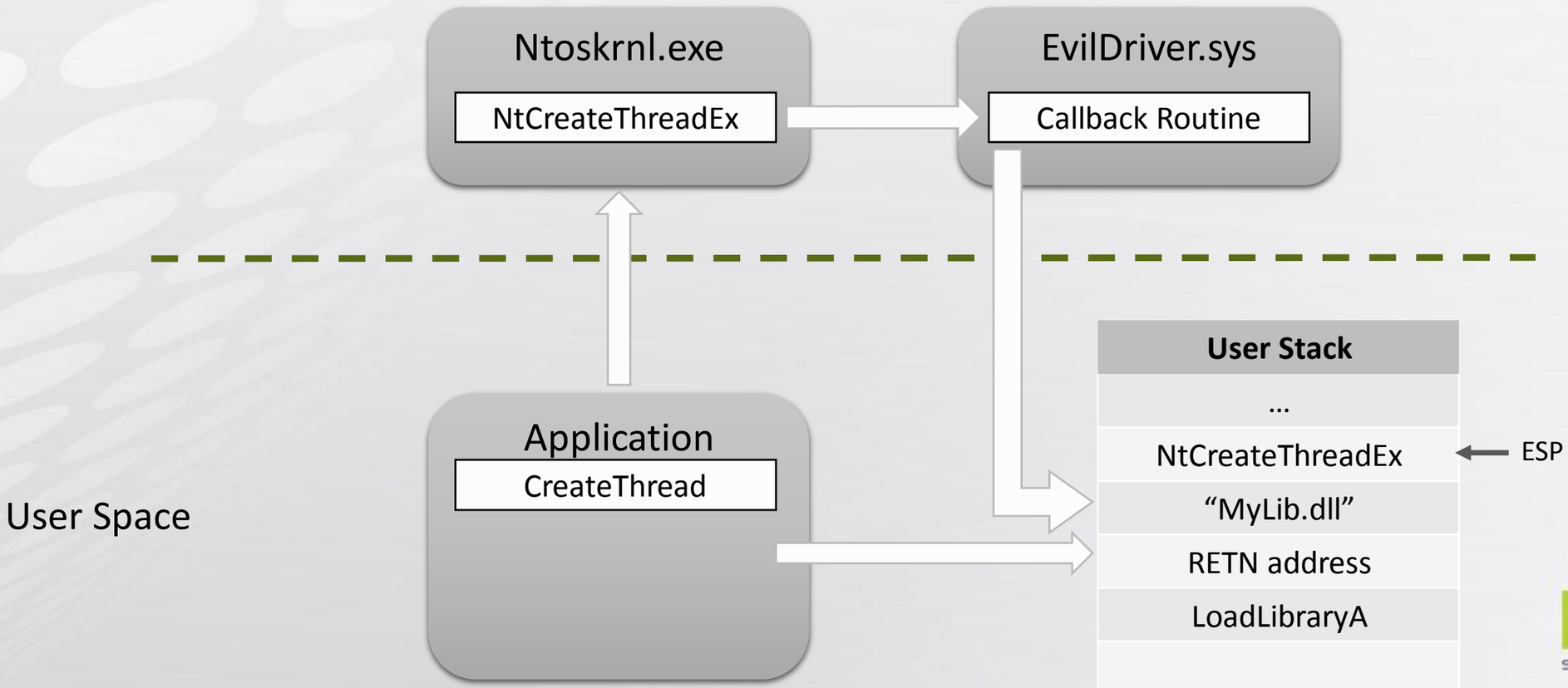
- Wait for some callback that runs in context of a system call (Thread Creation, Registry Access,…)



ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Codeless Code Injection

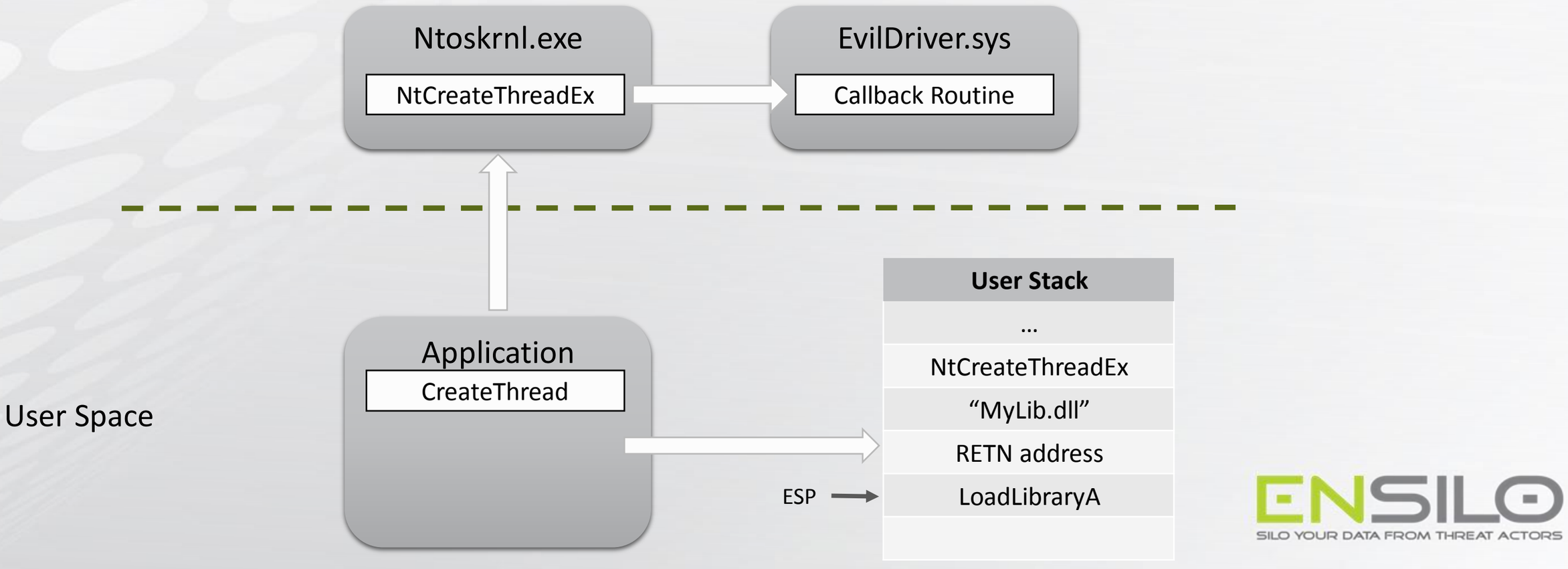
- Build ROP Chain on the user stack



ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Codeless Code Injection

- Set the stack pointer to the start of the ROP chain



Codeless Code Injection - Challenges

- Getting Return Values – how do we get return values back to kernel mode
  - Solution 1: Use device handle to get notifications (deep dive coming up)
  - Solution 2: Save return value somewhere and trigger some hook-able event (Such as registry access)
- Deadlocks – if the user mode caller holds a lock that our injected function needs
  - Solution: Use a dedicated thread
- Callback – What if we need to a user-mode callback function
  - Solution 1: Use a function that always triggers some hook-able event and fix the context
  - Solution 2: Just don't use such functions ☺

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Codeless Code Injection - NtClose as a callback

- Create a device using IoCreateDevice
- Whenever we require a callback, create a handle for the device in the target process
- Build the following ROP chain:

User Stack
…
Device Handle
RETN address
NtClose
MOV [EDX], EAX Gadget
Writable Location
POP EDX Gadget
…

ESP →

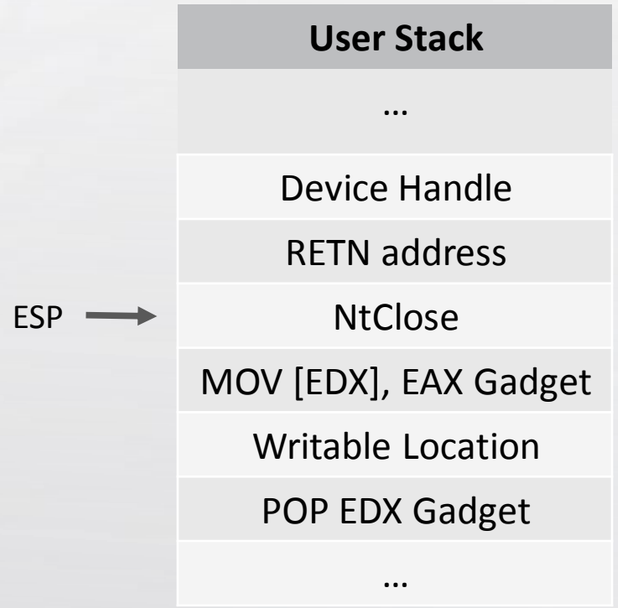
=

MOV EDX, ADDRESS
MOV [EDX], EAX
NtClose(DeviceHandle)

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Codeless Code Injection - NtClose as a callback

- When NtClose(Device Handle) is executed then the IRP_MJ_CLEANUP handler will be called
- The required data will reside in the target address



* moving EAX to some register is not safe because of WOW64

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

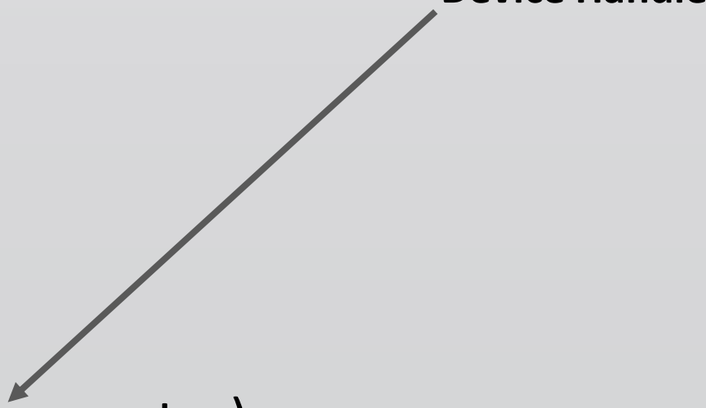
Codeless Code Injection – Creating dedicated thread

```
HANDLE WINAPI CreateThread(
  _In_opt_  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_      SIZE_T                 dwStackSize,
  _In_      LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt_  LPVOID                 lpParameter,
  _In_      DWORD                  dwCreationFlags,
  _Out_opt_ LPDWORD                lpThreadId
);

DWORD WINAPI ThreadProc(_In_ LPVOID lpParameter);
```

NtClose

Device Handle



Codeless Code Injection – Creating a Dedicated Thread

- Wait for some callback that runs in context of a system call (Thread Creation, Registry Access,…)
- Build the following ROP Chain

User Stack
…
0
0
Device Handle
NtClose
0
NULL
RETN address
ESP → CreateThread
…

=

```
CreateThread(NULL,
             0,
             NtClose,
             Device Handle,
             0,
             0);
```

- When Handle cleanup is triggered start injecting ROP chains to the new thread

Codeless Code Injection – API

```
ROProgram *CreateProgram(CHAR* TargetName)–
```
Creates a ROP program that's to be run in a target process

```
AddStep(ROProgram* Program, ROProgramStepProc StepProc)–
```
Adds a step to the program. A Single ROP chain that ends in callback(NtClose)

```
RunProgram(ROProgram* Program) –
```
Wait's for the target process to create a thread and starts a new dedicated thread

```
UserLibrary OpenUserLibrary(CHAR* LibName)–
```
Finds a DLL in the target process memory

```
Call(ROPChain* RopChain, UserLibrary Module, CHAR* Function, u32 NumberOfArgs, ...) –
```
Add a call to a ROP chain

```
LoadLibrary(ROPChain* RopChain, char* Name) –
```
Load a Library into the target process

```
WriteToUser(ROPChain* RopChain, u8* Buf, u32 BufSize) –
```
Writes a buffer of data to user-mode

ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Codeless Code Injection – API Example

Load a new library to the target process:

```
ROProgram* Program = CreateProgram("iexplore.exe“);
AddStep(Program, LoadSomeLibrary);
RunProgram(Program);

…
bool LoadSomeLibrary(ROProgram* Program, ROPChain* Chain){
     return LoadLibrary(Chain, “MyLibrary.dll”);
}
```

Demo

Reverse Shell from WINLOGON

Summary

- Code injection remains an important capability for both malware and security/software vendors
- Like exploits, techniques are becoming less generic (PowerLoader)
- New User and Kernel injection techniques are constantly being invented
- We can probably expect new advanced injection methods in the near future
- Slides and Source-code will be available on BreakingMalware in a few days

Thank You!

www.enSilo.com @enSiloSec Company/enSilo ensilo.com/blog contact@ensilo.com