Draw me a Local Kernel Debugger

Samuel Chevet & Clément Rouault

21 October 2015

Where does this talk come from?

BTF (Single-step on branches)

- `nt!KiSaveProcessorControlState`
- This feature seems not supported anymore on new CPU
- We wanted to be able to use this feature on our new CPU (not amd64)

BTS (Branch Tracing Store)

- `nt!VfInitializeBranchTracing`
- Partially implemented, could be nice to have a working POC

Where does this talk come from?

- We looked at the options to achieve that
- We started looking at WinDbg
- We wanted easier scriptability
- We looked at how WinDbg works
- So . . .

Let's draw a Local Kernel Debugger

Agenda

- Windows local kernel debugging
- DbgEngine for dummys
- Python kungfu
- Demo

Agenda

1 Introduction

Windows kernel debugging

Use case of kernel debugging

- Reverse engineering
  - Understand (hidden) features
  - Study patch Tuesday
  - Hunt vulnerabilities
- Exploit development
- Driver development
- Low level interaction

Windows kernel debugging

Debug settings

- Network cable
- USB (3.0 / 2.0)
- Serial cable
- Serial over USB
- **Locally**

Windows **local** kernel debugging

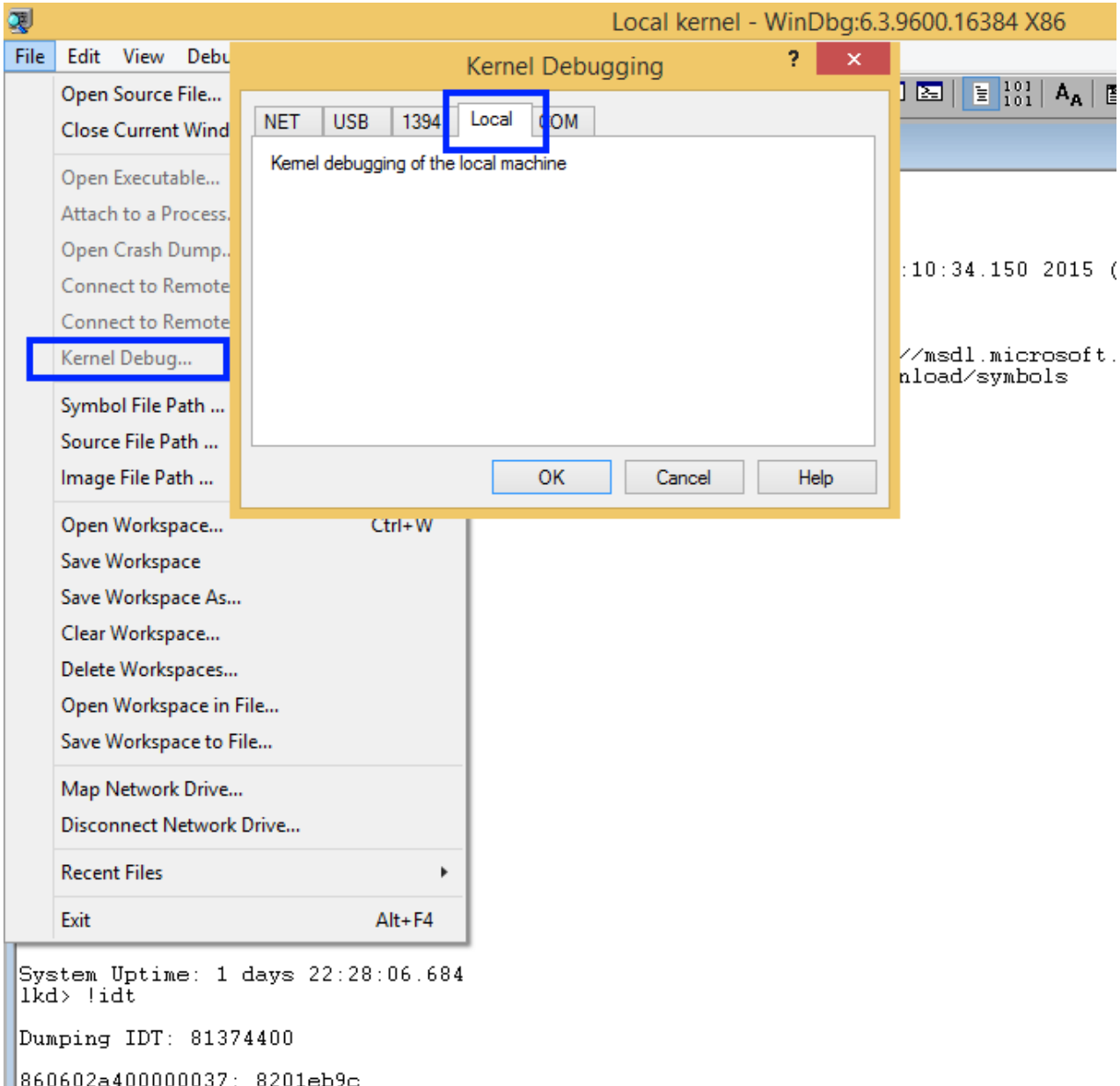
Locally?

- "Debugger" runs on the same computer
- Dump memory
  - Data structure used by processor (GDT, IDT, . . .)
  - Windows internal structures
  - Process list, handles, . . .
- Modify memory, I/O, MSRs
  - Enable hidden features
  - Fix bugs ☺

Windows **local** kernel debugging

Samuel Chevet & Clément Rouault

WinDbg allows to perform local kernel debugging



Windows **local** kernel debugging

Prerequisite

- Boot start options must be modified
- nt!KdDebuggerEnabled must be equal to 1
- "DEBUG" in

  `HKLM\System\CurrentControlSet\Control\SystemStartOptions`
- `bcdedit /debug on` || `msconfig.exe`

Agenda

2 DBGEngine

DBGEngine

- `WinDbg` uses `dbgEng.dll` : Debugger Engine
- Provides interfaces for examining and manipulating targets
- Can acquire targets, set breakpoints, monitor events, ...

Can we write our standalone Local Kernel Debugger?

Dissecting dbgeng.dll

dbgeng.dll

- Few exported functions (only one interesting)

```
HRESULT DebugCreate(__in REFIID InterfaceId, __out PVOID* Interface);
```

Creates a new **C**omponent **O**bject **M**odel (COM) interface of type `IDebugClient`

IDebugClient

Main object, queries other COM interfaces

- `IDebugControl`: Controls the debugger
- `IDebugSymbols`: Symbols stuff (`dbghelp.dll`, `symsrv.dll`)
- `IDebugDataSpaces`: Read / Write operations

Dissecting dbgeng.dll


```
HRESULT AttachKernel(
  [in]             ULONG Flags,
  [in, optional]   PCSTR ConnectOptions
);
```

IDebugClient (debugger.chm)

```
// Attach to the local machine.  If this flag is not set
// a connection is made to a separate target machine using
// the given connection options.
#define DEBUG_ATTACH_LOCAL_KERNEL      0x00000001
```

dbgeng.h

Not documented inside MSDN nor debugger.chm

Dissecting dbgeng.dll

- If we try to call the method, we end up in `dbgeng!LocalLiveKernelTargetInfo::InitDriver`
- This function checks if the current process name is `WinDbg` / `kd`
- If TRUE, it extracts a signed driver (`kldbgdrv.sys`) from the binary's resources
  - `lpName` = 0x7777
  - `lpType` = 0x4444

Dissecting kldbgdrv.sys

kldbgdrv.sys

- Create a device \\.\kldbgdrv
- Wrapper around nt!KdSystemDebugControl via DeviceIoControl (dwIoControlCode = 0x22C007)

nt!KdSystemDebugControl

- Check the value of nt!KdDebuggerEnabled (set during system startup)
- Read/Write: I/O, Memory, MSR, Data Bus, KPCR, ...
- nt!KdpSysReadIoSpace & nt!KdpSysWriteIoSpace broken, allows only aligned I/O

Stand-Alone application

Custom LKD

- Use `dbgeng.dll` like `WinDbg`
- Put `kldbgdrv.sys` inside our own resources

```
> type poc.rc
0x7777          0x4444      "dep\\kldbgdrv_64.sys"
> rc.exe /nologo poc.rc
```

- Add 3 others resources
  - `dbgeng.dll`
  - `dbghelp.dll`
  - `symsrv.dll`

No need to install anything ☺

Stand-Alone application

- Name our executable `WinDbg.exe` / `kd.exe` or hook `kernel32!GetModuleFileNameW`
- Enable SeDebugPrivilege / SeLoadDriverPrivilege
- Check if `debug` mode is enable
- Load `dbgeng.dll` (from extracted resources)
- Create an `IDebugClient` and `IDebugControl` interface with `DebugCreate`
- Call `AttachKernel` with DEBUG_ATTACH_LOCAL_KERNEL
- Call `WaitForEvent` until debugger is attached

Agenda

3 Python

What we need

Problems

- Call COM interface in Python
- `kernel32!GetModuleFileNameW` must return `windbg.exe`
- Embed `kldbgdrv.sys` as a resource

What we need

Problems

- Call COM interface in Python
- `kernel32!GetModuleFileNameW` must return `windbg.exe`
- Embed `kldbgdrv.sys` as a resource

Solutions

- `ctypes` module
- **I**mport **A**ddress **T**able (IAT) hooks

COM with ctypes


```
/* The SetSymbolPath method sets the symbol path. */
HRESULT SetSymbolPath(
  [in] PCSTR Path
);
```

```
int __stdcall IDebugSymbols::SetSymbolPath(PVOID, LPCSTR)
```

HOWTO

```
# SetSymbolPath is the 42nd entry in IDebugSymbols's vtable
SetSymbolPathFunction = WINFUNCTYPE(HRESULT, c_char_p)(41, "SetSymbolPath")
SetSymbolPathFunction(DebugSymbolsObject, "C:\\whatever")
# Abstract stuffs
kdbg.DebugSymbols.SetSymbolPath("C:\\symbols")
```

IAT hooks in Python

Steps

- Find the IAT entry (PEB + PE Parsing)
- Hook it with a stub able to call our Python function

What we need

- Python → native execution
- native execution → Python

ctypes magic once again


```python
def get_peb_addr():
    # mov    rax,QWORD PTR gs:0x60; ret
    get_peb_64_code = "65488B042560000000C3".decode("hex")
    # Declare a function type that takes 0 arg and returns a PVOID
    func_type = ctypes.CFUNCTYPE([PVOID])
    addr = write_code(get_peb_64_code)
    # Create a function of type 'func_type' at addr
    get_peb = func_type(addr)
    # Call it
    return get_peb()
```

Python → Native execution

Samuel Chevet & Clément Rouault

ctypes magic once again


```
def my_callback(x, y):
    "Do whatever you want"
    return 0

# Create the type of the function
func_type = WINFUNCTYPE(c_uint, c_uint, c_uint)
# c_callable contains a native stub able to transform
# the arguments to Python object and call Python code
c_callable = func_type(my_callback)
```

Native execution → Python

- This stub is not enough for our IAT hook as we need to prepare threads to call Python code
- *Manually* create another stub that will call the ctypes stub

Additional work

Make threads able to execute Python code

- Need to call `PyGILState_Ensure` before the `ctypes` stub and `PyGILState_Release` after
- Need to leave registers and stack untouched for proper arguments parsing
  - Poping and saving the return address elsewhere
  - Need to save registers (not on the stack)

HIDE ALL THE MAGIC !

Callback decorator

- Create a wrapper function that:
  - Handle all the low level magic
  - Create a Python function calling the real API
  - Call our hook with original arguments

```
@Callback(ctypes.c_void_p, ctypes.c_ulong)
def exit_callback(x, real_function):
    print("Try to quit with {0}".format(x))
    if x == 42:
        print("TRYING TO REAL EXIT")
        return real_function(1234)
    return 0x4242424243444546

exit_process_iat.set_hook(exit_callback)
```

Bonus

We can generate specialized Callback decorators for functions with known arguments

Application to dbgeng.dll

- dbgeng!LocalLiveKernelTargetInfo::InitDriver checks the name of the current process

```
@windows.hooks.GetModuleFileNameWCallback
def EmulateWinDbgName(hModule, lpFilename, nSize, real_function):
    if hModule is not None:
        return real_function()
    ptr_addr = ctypes.cast(lpFilename, ctypes.c_void_p).value
    v  = (c_char * 100).from_address(ptr_addr)
    path = "C:\\windbg.exe"
    path_wchar = "\x00".join(path) + "\x00\x00\x00"
    v[0:len(path_wchar)] = path_wchar
    return len(path_wchar)
```

Hook for kernel32!GetModuleFileNameW

Application to dbgeng.dll

- Embed `kldbgdrv.sys` as a resource (0x7777, 0x4444)

```
DRIVER_RESOURCE = Resource(DRIVER_FILENAME, 0x7777, 0x4444)

@windows.hooks.LoadResourceCallback
def LoadResourceHook(hModule, hResInfo, real_function):
    if hResInfo in HRSRC_dict:
        return HRSRC_dict[hResInfo].load_resource()
    return real_function()

# Simplified implementation of Ressource.load_resource
# Real implementation must keep driver_data alive so it's
# not garbage collected
def load_resource(self):
    driver_data = open(self.filename, 'rb').read()
    char_p = ctypes.c_char_p(driver_data)
    real_addr = ctypes.cast(char_p, ctypes.c_void_p).value
    return real_addr
```

Hook for `kernel32!LoadResource`

Results


```python
from dbginterface import LocalKernelDebugger

kdbg = LocalKernelDebugger()
addr = kdbg.get_symbol_offset("nt!KiSystemStartup")
print("nt!KiSystemStartup -> " + hex(addr))
data =  kdbg.read_virtual_memory(addr, 0x10)
print("Read 0x10 at symbol :\n" + repr(data))
```

Python LKD in action

```
> python64 test.py
nt!KiSystemStartup -> 0xffffffff81081310L
Read 0x10 at symbol :
'U\x8b\xec\x83\xec \x8b]\x08\x89\x1dhD\x07\x81\x8b'
```

Output

Agenda

4 Level UP

Limitations

- Impossible to perform non-aligned I/O using (`nt!KdpSysReadIoSpace` & `nt!KdpSysWriteIoSpace`)
- Unable to allocate kernel memory
- Unable to call custom kernel functions

Upgrade

- We didn't want to disable Secure Boot
- We didn't want to rely on compilation step

Upgrade

- We didn't want to disable Secure Boot
- We didn't want to rely on compilation step

Solution

- Use `kldbgdrv` driver features to upgrade it
- Add new execution path during IOCTL handling

Upgrade

- We can now "register" custom code execution to custom IOCTL code

Features

- Perform non-aligned I/O
- Call custom kernel functions with arguments
- Allocate kernel memory (and map it to user-land)

Upgrade Example


```
    self.upgrade_driver_add_new_ioctl_handler(DU_MEMALLOC_IOCTL,
            Alloc_IOCTL.get_code())


    # Wrapper in LocalKernelDebugger
    @require_upgraded_driver
    def alloc_memory(self, size=0x1000, type=0, tag=0x45544942):
        buffer = struct.pack("<QQQ", type, size, tag)
        res = c_uint64(0x44444444)
        DeviceIoControl(handle, DU_MEMALLOC_IOCTL, buffer,
                                len(buffer),  byref(res), sizeof(res))
        return res.value
```

Memory allocation upgrade

Upgrade Example


```
>>> hex(kdbg.alloc_memory(0x42000, type=0, tag=0x21444b4c))
'0xffffe001572b2000L'
```

Kernel memory allocation from Python

```
lkd> !pool 0xffffe001572b2000
Pool page ffffe001572b2000 region is Unknown
*ffffe001572b2000 : large page allocation, Tag is LKD!, size is 0x42000 bytes
            Owning component : Unknown (update pooltag.txt)
```

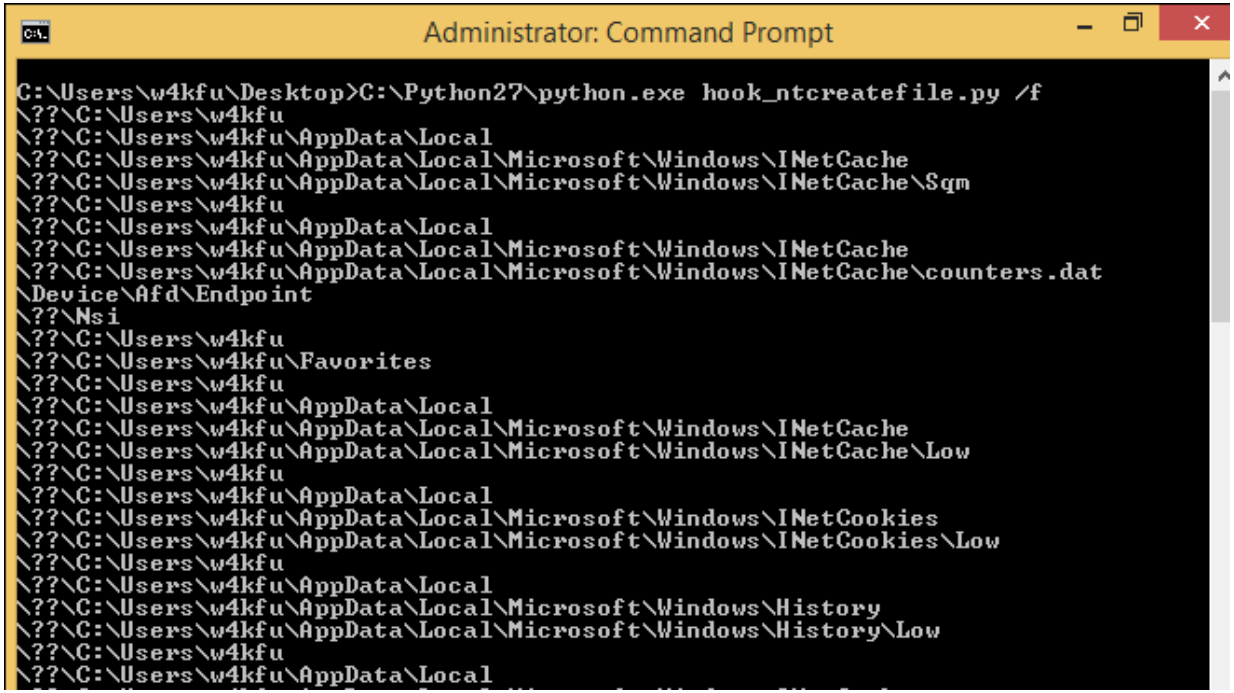
Proof of work

Agenda

5 Demo

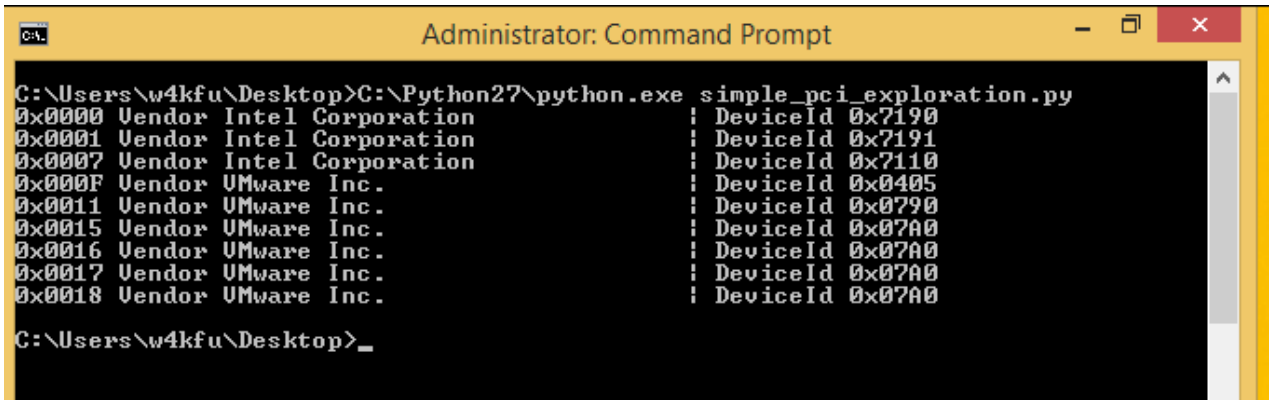
Demo

- Setup Inline hook on `nt!NtCreateFile`



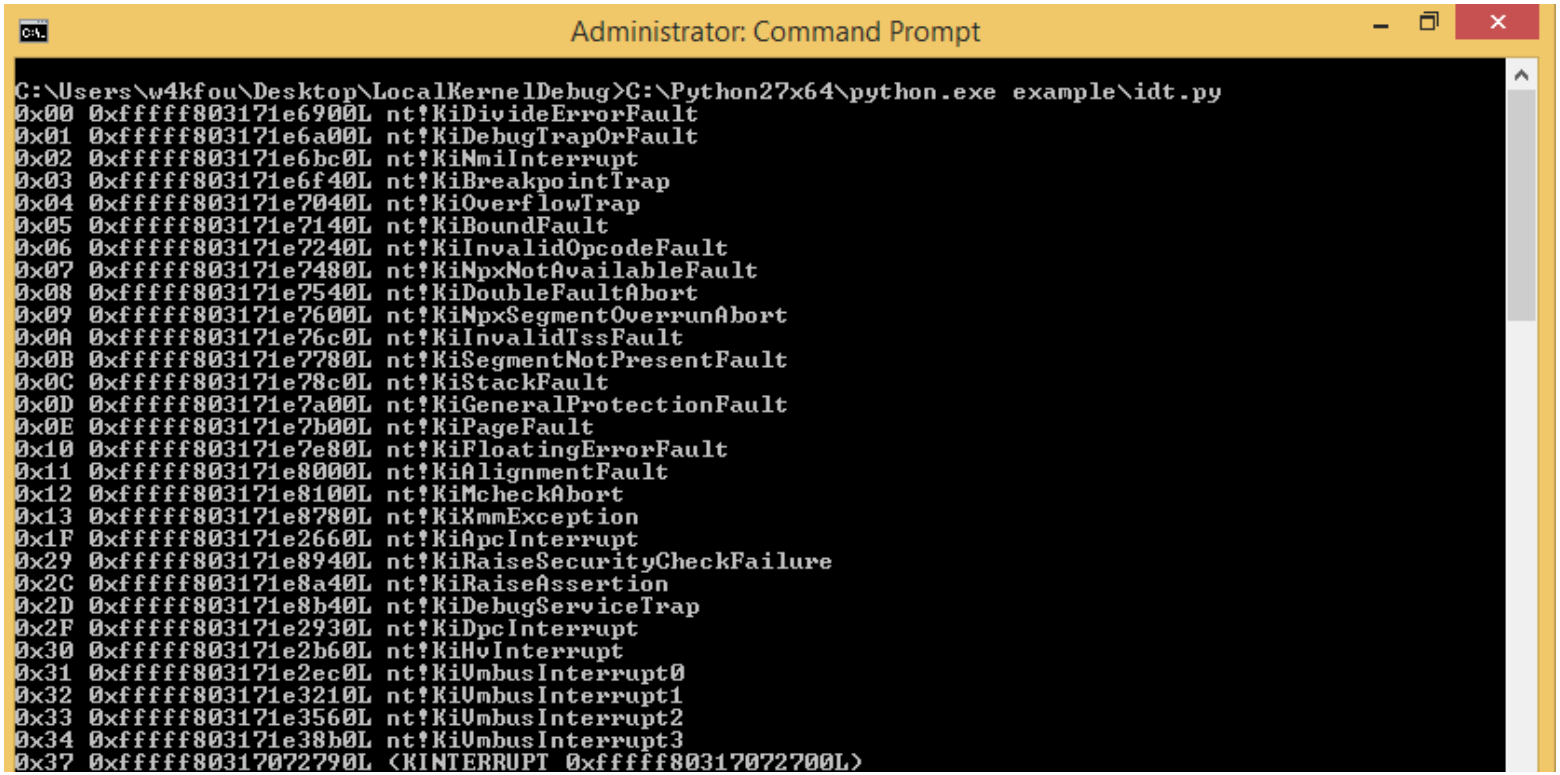
Demo

- Display devices attached to PCI bus
- `DebugDataSpaces::ReadBusData`



Demo

- Display the interrupt dispatch table and `KINTERRUPT` associated



Demo

Branch Trace Store (BTS)

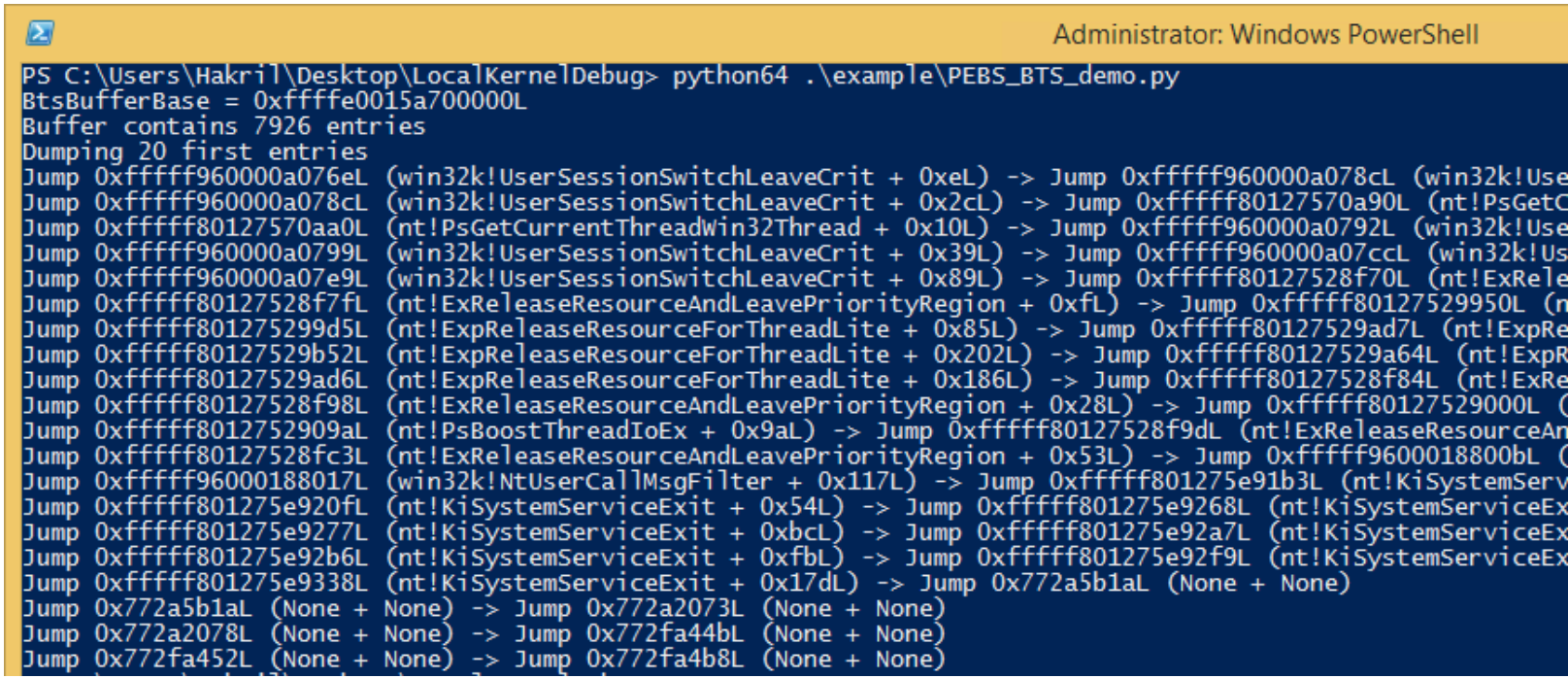
- Store all the branches (src and dst) taken on a CPU to a buffer
- IA32_DEBUGCTL_MSR, MSR_IA32_DS_AREA
- ...

HowTo

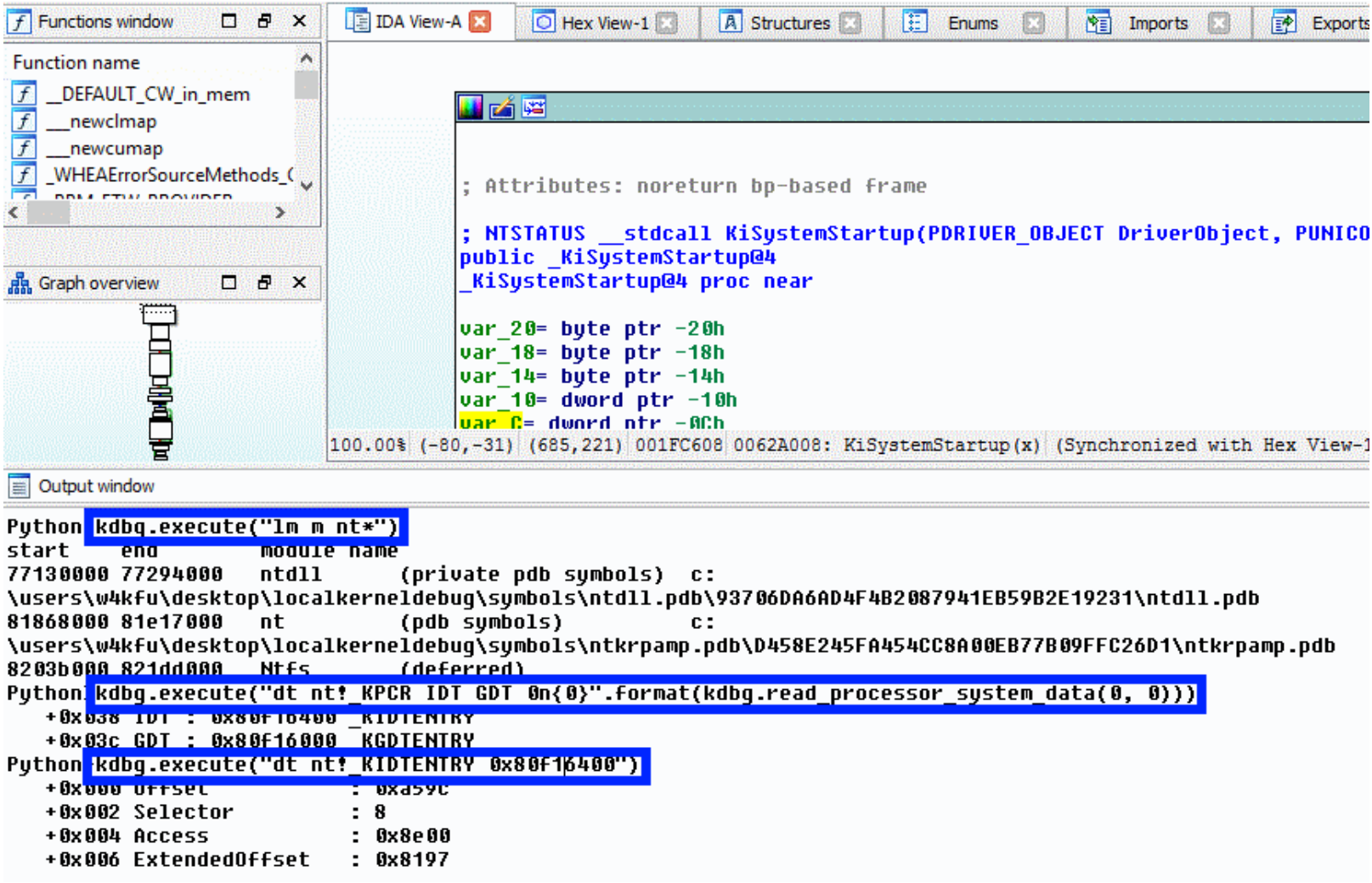
- Setup the Debug Store (DS) Area
- Setup the BTS related fields in DS
- Activate BTS (bit 6 & 7 IA32_DEBUGCTL_MSR)

Demo

Samuel Chevet & Clément Rouault



Demo



Samuel Chevet & Clément Rouault

Agenda

6 Conclusion

Conclusion

- **L**ocal **K**ernel **D**ebugging is a really nice feature provided by the Windows kernel
- Such scriptability in python from user-land can be interesting in many use-cases that we are still exploring
- Source code available at `https://github.com/sogeti-esec-lab/LKD`

Questions?

Thank you for your attention

@w4kfu
@hakril