WAVESTONE

DEF CON 30 – DemoLabs

EDR detection mechanisms and bypass techniques with EDRSandBlast

Maxime MEIGNAN (th3m4ks) & **Thomas DIOT (_Qazeer)**

Hi !

Who are we ?

- Thomas Diot (_Qazeer)
- Maxime Meignan (th3m4ks)

@ Wavestone

Why EDRSandblast ?

- EDRs are more and more prevalent in corporate environments
- EDRs may need to be bypassed in red-team engagements, as well as during pentests

Hi !

github.com/wavestone-cdt/EDRSandblast

What is EDRSandblast ?

- Tool written in C
- Detects common monitoring techniques used by EDR software on Windows endpoints
- Implements techniques to bypass them (both user-land and kernel-land)
- Exists as a CLI tool and as a static library to include in another project

/ 01 So you want to dump LSASS ?

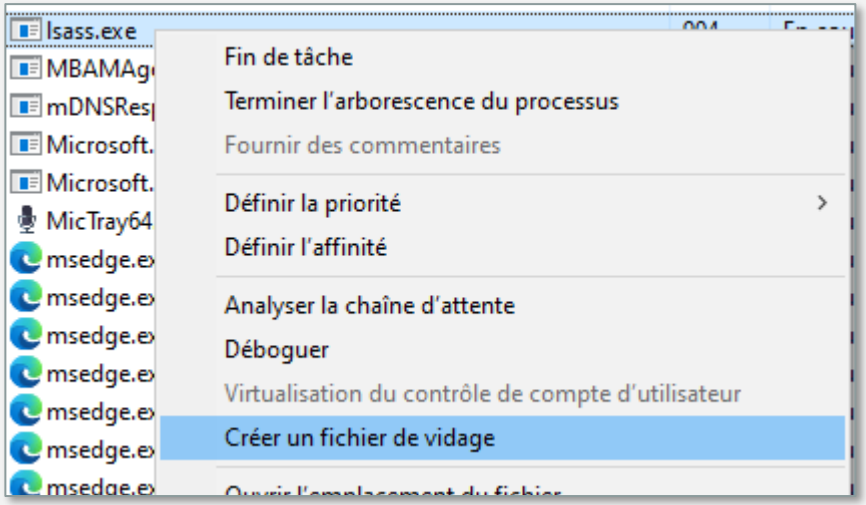
With the tool of your choice

```
C:\no_scan\programs\SysInternals>procdump.exe -ma 904 lsass.dmp
```

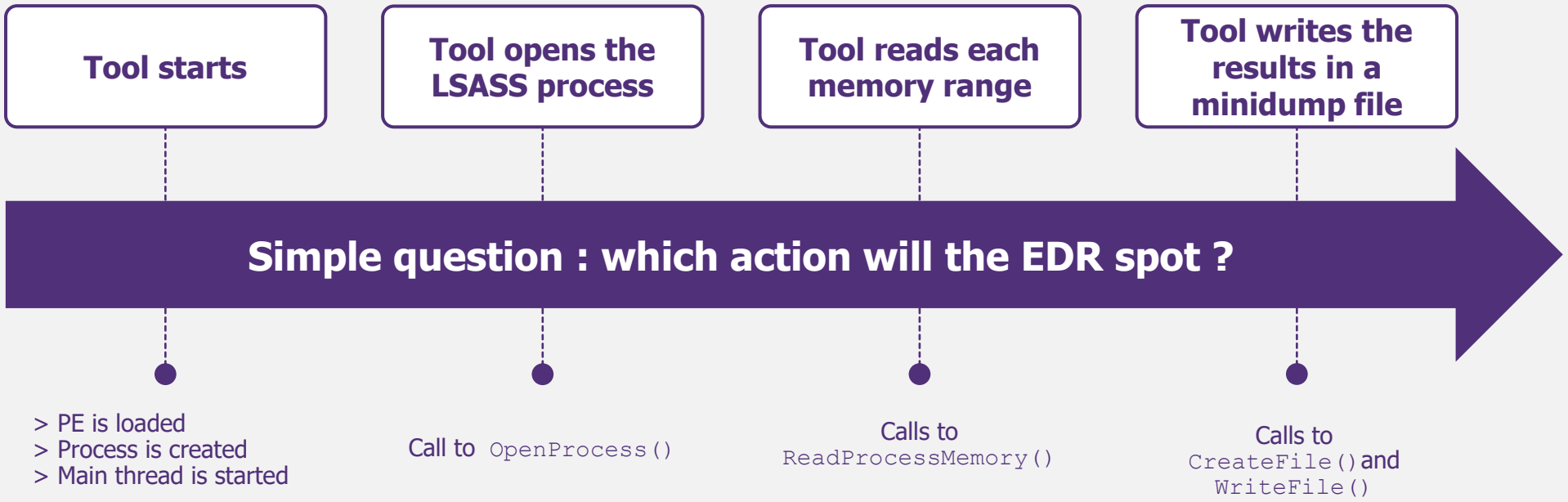
```
C:\Windows\System32>rundll32.exe comsvcs.dll MiniDump 904 lsass.dmp full
```

```
  .#####.   mimikatz 2.2.0 (x64) #19041 Sep 18 2020 19:18:29
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz # sekurlsa::logonPasswords
```



What happens classically during a process dumping



Easy answer: the EDR saw you at every step

How come the EDR knows everything ?

- EDR registered **callback functions** with `PsSet{CreateProcess,CreateThread,LoadImage}NotifyRoutine()`
- EDR's driver is **notified** by the kernel at each **process creation**, **thread creation**, or **PE loading** (executable, library, driver)

Kernel notify routine callbacks allow EDRs to be notified of process or thread creation and image loading

- The **Kernel notify routine callbacks** are added through documented APIs to define **driver-supplied callback routines**.
  The callbacks routines are then stored in undocumented arrays in kernel memory: ***PspCreateProcessNotifyRoutine***, ***PspCreateThreadNotifyRoutine***, and ***PspLoadImageNotifyRoutine***

- The **callback routines** are then **called** upon the **occurrence of their associated system events**.

```
// Process creation callbacks.
void PcreateProcessNotifyRoutine(HANDLE ParentId, HANDLE ProcessId, BOOLEAN Create);
// PS_CREATE_NOTIFY_INFO contains information about the created process (PPID, image and CLI notably).
void PcreateProcessNotifyRoutineEx(PEPROCESS Process, HANDLE ProcessId, PPS_CREATE_NOTIFY_INFO CreateInfo);

// Thread creation callbacks
void PcreateThreadNotifyRoutine(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Created);

// Image loading callbacks
// IMAGE_INFO contains information about the loaded image (signature level / type, size, ...).
void PloadImageNotifyRoutine(PUNICODE_STRING FullImageName, HANDLE ProcessId, PIMAGE_INFO ImageInfo);
```

Prototypes the Kernel notify routine callbacks must follow

Demo

How come the EDR knows everything ?

- Using these notifications, EDR may also **insert its own libraries** inside each process memory space before it starts

Demo

How come the EDR knows everything ?

- EDR registered **callback functions** with `ObRegisterCallbacks()`
- EDR's driver is **notified** by the kernel at each **handle** creation or duplication on **threads** or **processes**
- EDR can **monitor** `OpenProcess()` calls and even **block the handle opening**

ObRegisterCallbacks allows EDRs to be notified of handle operations by processes and threads

/ The **Kernel Object callbacks** are added through a documented API to define **driver-supplied *ObjectPreCallback*** and ***ObjectPostCallback* routines**.
The callbacks routines are then stored in an **undocumented doubly linked list**, with no symbols.

/ The **callback routines** are then **called when** or **after a process** or **thread make a handle operation**.

```
void PobPreOperationCallback(PVOID RegistrationContext, POB_PRE_OPERATION_INFORMATION OperationInformation);

void PobPostOperationCallback(PVOID RegistrationContext, POB_POST_OPERATION_INFORMATION OperationInformation);
```

Prototypes the Kernel ObjectPreCallback and ObjectPostCallback routines must follow

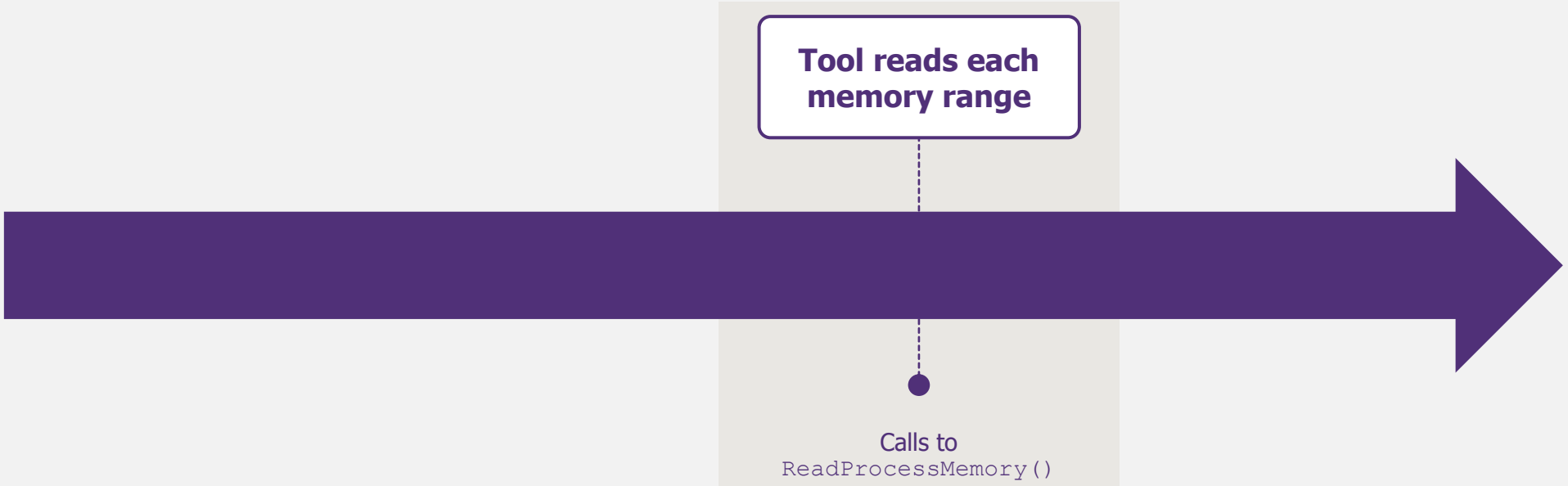
The ***OB_PRE_OPERATION_INFORMATION*** and ***OB_POST_OPERATION_INFORMATION*** contain information about the operation and notably:

- The target of the handle operation
- The desired / granted access (as an ACCESS_MASK)

```
typedef struct _OB_PRE_CREATE_HANDLE_INFORMATION {
  ACCESS_MASK DesiredAccess;
  ACCESS_MASK OriginalDesiredAccess;
} /* [...] */;
```

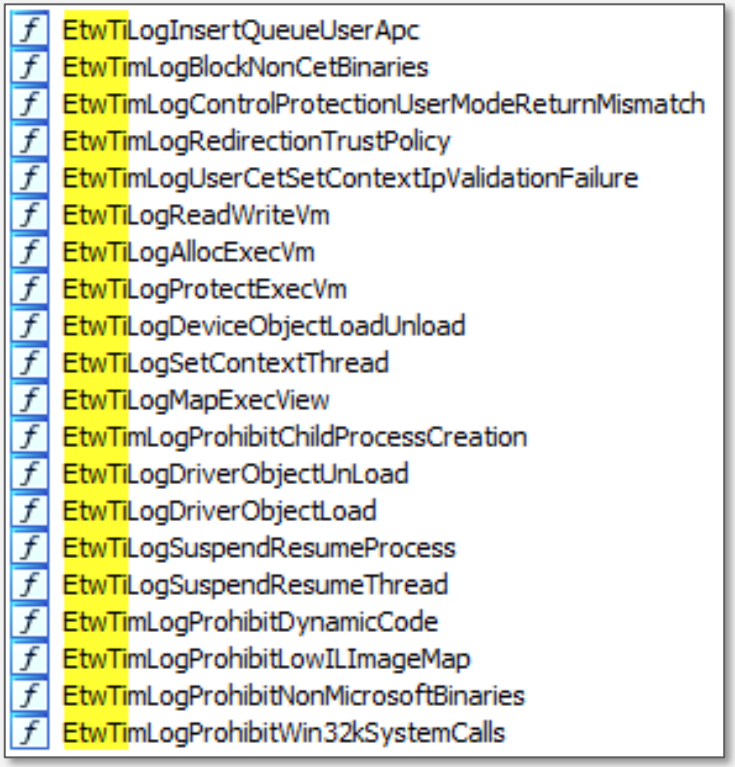
Demo

How come the EDR knows everything ?



- EDR **subscribed** to a special **event provider** called *ETW Threat Intelligence,* reserved to security products (signed as « *Early-Launch-Antimalware* »)
- This provider resides in **kernel memory** and cannot be altered from userland
- Calling certain kernel functions (ex. `MiReadWriteVirtualMemory`) will generate events available for the EDR to analyze

EDRs can subscribe to the ETW Microsoft-Windows-Threat-Intelligence provider to receive telemetry on Windows API usage from the kernel



List of ETW TI functions in a recent Windows build

```
loc_1405EA827:                          ; CODE XREF: MiReadWriteVirtualMemory+170↑j
                mov     [rsp+28h], rsi
                mov     [rsp+20h], r13
                mov     r9d, r12d
                mov     r8, r14
                mov     rdx, r10
                mov     ecx, edi
                call    EtwTiLogReadWriteVm
                jmp     short loc_1405EA7D2
```

Example of a call to the ETWTI logging function in nt!MiReadWriteVirtualMemory

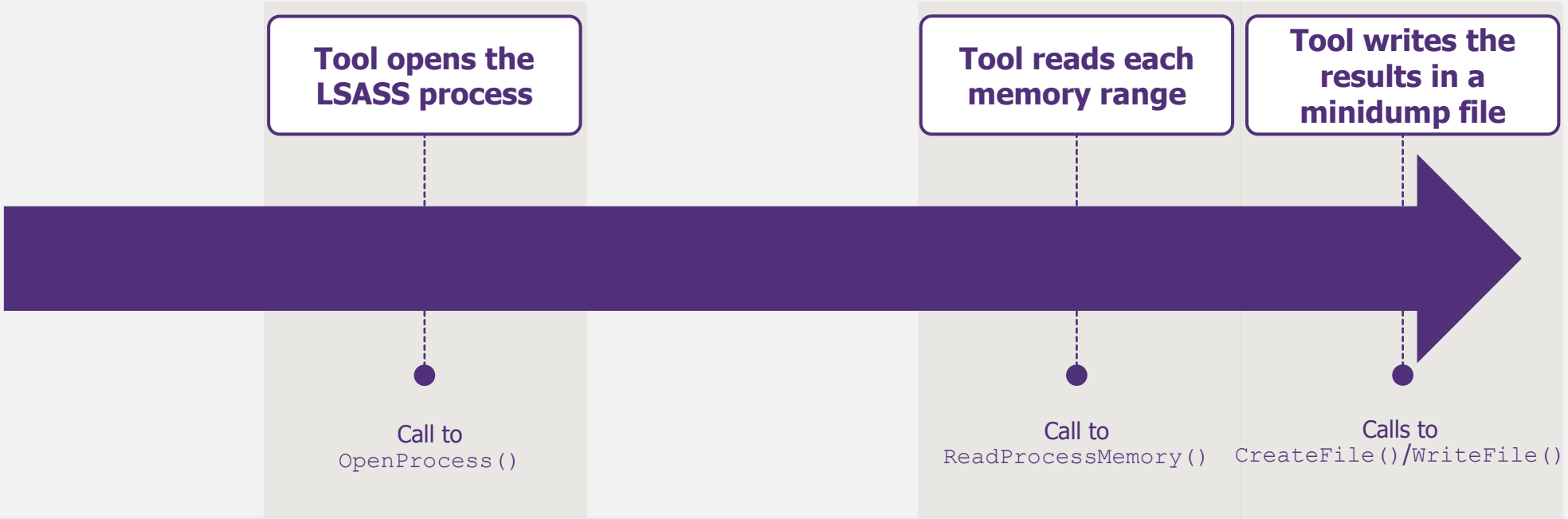
```
<template tid="KERNEL_THREATINT_TASK_READVMArgs_V1">
  <data name="OperationStatus" inType="win:UInt32" />
  <data name="CallingProcessId" inType="win:UInt32" />
  <data name="CallingProcessCreateTime" inType="win:FILETIME" />
  <data name="CallingProcessStartKey" inType="win:UInt64" />
  <data name="CallingProcessSignatureLevel" inType="win:UInt8" />
  <data name="CallingProcessSectionSignatureLevel" inType="win:UInt8" />
  <data name="CallingProcessProtection" inType="win:UInt8" />
  <data name="CallingThreadId" inType="win:UInt32" />
  <data name="CallingThreadCreateTime" inType="win:FILETIME" />
  <data name="TargetProcessId" inType="win:UInt32" />
  <data name="TargetProcessCreateTime" inType="win:FILETIME" />
  <data name="TargetProcessStartKey" inType="win:UInt64" />
  <data name="TargetProcessSignatureLevel" inType="win:UInt8" />
  <data name="TargetProcessSectionSignatureLevel" inType="win:UInt8" />
  <data name="TargetProcessProtection" inType="win:UInt8" />
  <data name="BaseAddress" inType="win:Pointer" />
  <data name="BytesCopied" inType="win:Pointer" />
</template>
```

Example an event fields generated by EtwTiLogReadWriteVm for virtual memory read operations

How come the EDR knows everything ?

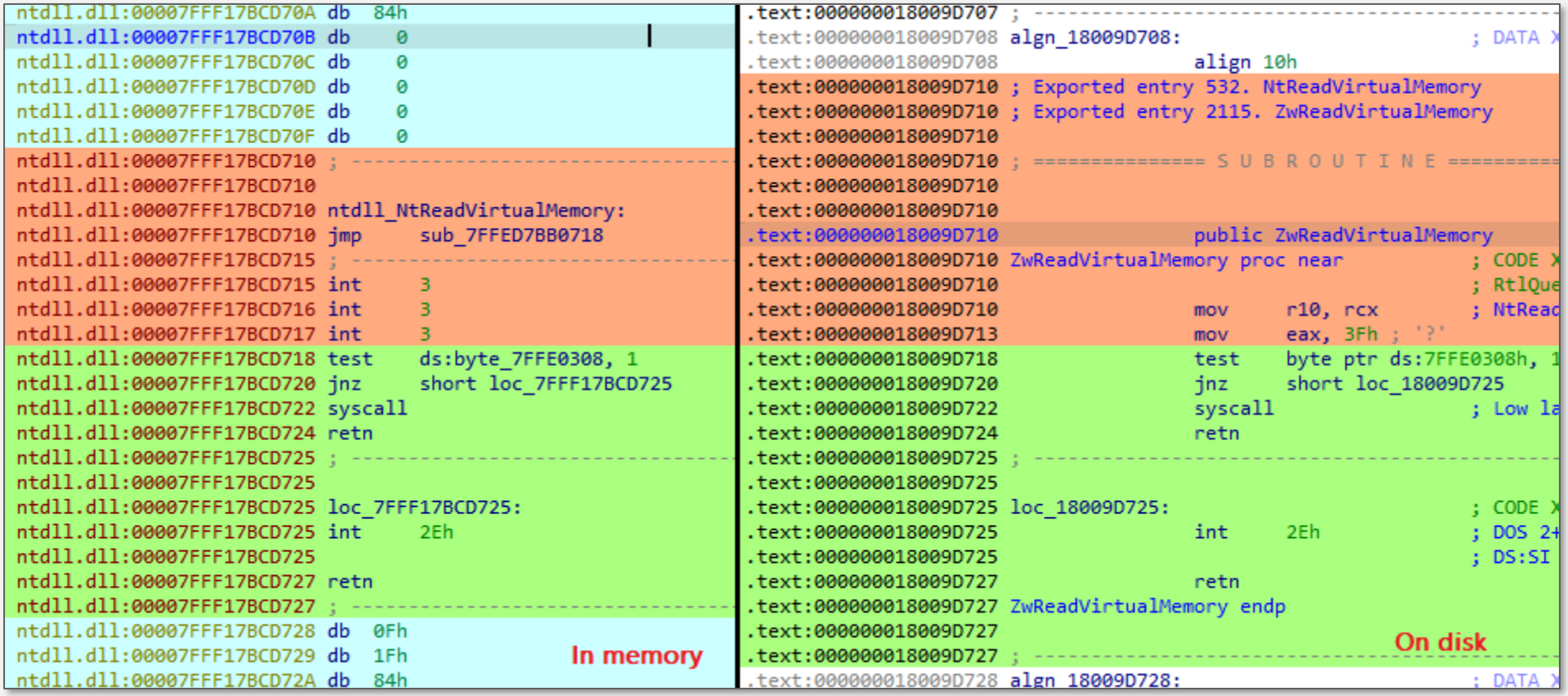
- EDR registered a ***minifilter driver*** with `FltRegisterFilter()`
- This driver will be called **each time** an I/O is performed on the **file-system**
- This allow the EDR to **intercept file creations** and scan their content

How come the EDR knows everything ?



- EDR **loaded its own library** at process start, remember ?
- The library installed **hooks** on all interesting userland functions for monitoring purposes
- At each (naive) call to a monitored function, the EDR will **inspect arguments** or **return values** to detect « malicious actions »

Example of a hook installed by the EDR



Example of a hook in the ntdll.NtReadVirtualMemory function introduced by an EDR

/ 02 How to bypass these monitoring techniques

Hooks are detected and removed by leveraging on-disk DLLs

Detecting hooks

For all loaded DLLs of a process, the content **on disk** is compared to the one **in memory**. Every difference found in a code section is a potential hook.

Removing hooks

Instructions overwritten by hooks are **restored** using the **on-disk** content. Page containing the instructions is temporarily set to be **writable** using `NtProtectVirtualMemory`.
However, this function is probably hooked itself by the EDR.

Multiple techniques are implemented to get an unmonitored call to any hooked function, like NtProtectVirtualMemory

1. **Construct an unhooked** `NtProtectVirtualMemory` by allocating an executable trampoline **jumping over the hook**

2. **Search** and **use an existing trampoline allocated by the EDR itself** to get an unhooked version of `NtProtectVirtualMemory`

3. **Load an additional version of ntdll library** into memory and use the `NtProtectVirtualMemory` from this library

4. Use a **direct syscall to call** `NtProtectVirtualMemory`

Demo

Removing Kernel-land monitoring requires to be able and to know where to write in the kernel memory

Reading / writing kernel memory

- A **driver** can be leveraged to **access the kernel memory** as they **share the same memory address space**.
- Since the introduction of **Driver Signature Enforcement (DSE)**, new drivers (post 07/2015) must be certified by Microsoft Windows Hardware Quality Labs (WHQL).
- A legitimate and WHQL-certified but **vulnerable driver** can be exploited to obtain arbitrary read / write of kernel memory primitives.

Knowing where to write

- **Global variables' offsets** and **fields offsets in structures** are leveraged by EDRSanblast to know where to write (instead of relying on the search of memory patterns).
- Known offsets allow **more stability** and **reduce the risk of BSOD**.
- The offsets can be:
  - **Passed in a CSV file**, with 450+ versions of the Windows kernel supported to date
  - **Automatically recovered**, if the endpoint has Internet connectivity, **by downloading the .pdb** (from MS symbol server) associated with the targeted ntoskrnl version

EDRSanblast enumerates the routines registered with PsSet*NotifyRoutine or ObRegisterCallbacks and remove any callback routine linked to a predefined list of EDR drivers

Bypassing notify routine callbacks

Use offsets to the ***PspCreateProcessNotifyRoutine***, ***PspCreateThreadNotifyRoutine***, and ***PspLoadImageNotifyRoutine*** arrays to **iterate on the callbacks arrays** and **remove all callback functions pointing to an EDR driver memory space**.

Bypassing object callbacks

Uses offsets to the ***PsProcessType*** and ***PsThreadType*** global variables (_OBJECT_TYPE* structures) and the **CallbackList** field offset in theses structures to **retrieve the head of the ObRegisterCallbacks linked lists**.

Both lists are then walked and the ***PreOperation*** and ***PostOperation*** fields of the undocumented structure of each item are analyzed to **identify if the callbacks belong to an EDR** driver and to **disable the callback**, using the ***Enabled*** field.

The undocumented structure has been reversed and was constant from Windows 10 versions 10240 (July 2015) to 22000.

Demo

The ETW Microsoft-Windows-Threat-Intelligence provider can be disabled system-wide through a kernel arbitrary RW primitive

- **Patching a process memory** to disable user-land ETW loggers (for instance by patching ntdll!EtwEventWrite) **will not impact the ETW TI provider**.

  As can sometimes be incorrectly stated, process memory patching does not "Disable Event Tracing for Windows".

- **Disabling the ETW TI provider** with a kernel memory read/write primitive is simply a matter of **patching a value in the _ETW_GUID_ENTRY entry** representing the ETW TI provider in memory.

```
struct _ETW_GUID_ENTRY {
  _LIST_ENTRY          GuidList;              //0x00
  // ...
  _TRACE_ENABLE_INFO   ProviderEnableInfo;    //0x60
  // ...
};
```

```
struct _TRACE_ENABLE_INFO {
  ULONG     IsEnabled;                //0x00
  UCHAR     Level;                    //0x04
  UCHAR     Reserved1;                //0x05
  USHORT    LoggerId;                 //0x06
  ULONG     EnableProperty;           //0x08
  ULONG     Reserved2;                //0x0c
  ULONGLONG MatchAnyKeyword;          //0x10
  ULONGLONG MatchAllKeyword;          //0x18
};
```

Demo

github.com/wavestone-cdt/EDRSandblast

The vulnerable RTCore64.sys driver can be retrieved at:

https://tinyurl.com/Demo-RTCore64

Quick usage

EDRSandblast.exe <audit | dump | cmd | credguard | firewall> [--usermode] [--kernelmode]

Options

```
Actions mode:

        audit           Display the user-land hooks and / or Kernel callbacks without taking actions.
        dump            Dump the LSASS process, by default as 'lsass' in the current directory or at the
                        specified file using -o | --output <DUMP_FILE>.
        cmd             Open a cmd.exe prompt.
        credguard       Patch the LSASS process' memory to enable Wdigest cleartext passwords caching even if
                        Credential Guard is enabled on the host. No kernel-land actions required.

--usermode              Perform user-land operations (DLL unhooking).
--kernelmode            Perform kernel-land operations (Kernel callbacks removal and ETW TI disabling).
```

New features published this morning!

- **Object callbacks** detection and removal
- **Firewalling** of EDR components to block telemetry
- **Downloading and parsing** of the `ntoskrnl` PDB at runtime for offsets retrieval
- **Refactoring** of the kernel read/write primitives making the support of a new vulnerable driver simpler to implement
- Support of the **Dell vulnerable driver** `DBUtil_2_3.sys`
- Creation of a **simple API** to use EDRSandblast as a **static library**
- Implementation of a function that **returns a "safe" version of a hooked** Nt* function
- Implementation of an equivalent of `MiniDumpWriteDump` with **only Nt* functions** ("syscalls")

EDRSanblast can now be imported as a static library in your project to easily add EDR detection and bypasses capabilities

```
int main()
{
    EDRSB_CONTEXT ctx = { 0 };
    EDRSB_CONFIG cfg = { 0 };
    cfg.bypassMode.Usermode = TRUE;
    cfg.bypassMode.Krnlmode = TRUE;
    cfg.offsetRetrievalMethod.Internet = TRUE;
    cfg.offsetRetrievalMethod.File = TRUE;

    EDRSB_Init(&ctx, &cfg);
    Usermode_RemoveAllMonitoring(&ctx, EDRSB_UMTECH_Find_and_use_existing_trampoline);
    Krnlmode_RemoveAllMonitoring(&ctx);
    Action_DumpProcessByName(&ctx, L"lsass.exe", L"C:\\temp\\tmp.tmp", EDRSB_UMTECH_Find_and_use_existing_trampoline);
    Krnlmode_RestoreAllMonitoring(&ctx);
    EDRSB_CleanUp(&ctx);
}
```

Example of a simple LSASS dumper program that uses the EDRSandblast API

Any Questions, Suggestions, Ideas?

/ **04**

Annexes

The introduction of PatchGuard, to protects the Windows x64 kernel, forced security product vendors to adapt their detection mechanisms

/ **PatchGuard**, also known as Kernel Patch Protection (KPP), is a **protection mechanism for the Windows (x64) kernel memory** to prevent **illegitimate modifications of kernel memory**.

If an anormal modification is detected, **PatchGuard generates a "Bug Check"** (also known as "Blue Screen of Death").

No more **interceptions of syscalls** via **modifications of the System Service Descriptor Table (SSDT)** as the SSDT is a PatchGuard protected structure

Security products developers (and rootkits) **had to rethink their monitoring mechanisms** on 64-bit Windows OS.