Enumerating kernel notification callback routines, x64

This document apply to:

Windows 7,
Windows 7 SP1,
Windows 8,
Windows 8.1,
Windows 10 (TH1 up to 21H2)
Windows 11 (21H2 up to 23H2)

Table of Contents

1. Introduction

List of common kernel callback routines, their implementation and how to enumerate them.

Important note: since everything here related to kernel mode, a kernel mode driver is required for read memory operations.

Here and below kldbgdrv (WinDbg Kernel Local Debugging Driver) used. It provides stable memory read functionality implemented by KdSystemDebugControl ntos routine (requires Debug mode enabled and SeDebugPrivilege assigned). Any other own implemented driver can be used instead, e.g. rkhdrv50 or wodbgdrv.

Callbacks usually used by driver developers to gain notifications when certain events happen. That's why some of them often referenced as "notification routines" or "notifies" rather than "callbacks". Also you should distinguish callback routines with "Callback" object type that also play notification role but implemented as kernel mode object.

Common callbacks split on two sections: these callbacks that can only be used for notifications of events and these callbacks that can change event behavior.

Callbacks often abused by malicious/fraud/bloatware software because of their functionality.

There is no official way to enumerate these callbacks and their implementation is always undocumented and subject of change between Windows versions. Additionally proper enumeration require accessing to locks and other synchronization mechanisms that vary from one callback to other, they all not exported functions, variables and obviously cannot be called, accessed from user mode. We are reading this data from user mode with small chance of that incoherent data will be returned.

2. Callbacks Table

Routine Name	Minimum supported client[1]	Documentation status
PsSetCreateProcessNotifyRoutine	Windows 2000	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetcreateprocessnotifyroutine
PsSetCreateProcessNotifyRoutineEx	Windows Vista SP1	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetcreateprocessnotifyroutineex
PsSetCreateProcessNotifyRoutineEx2	Windows 10 (1703)	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetcreateprocessnotifyroutineex2
PsSetCreateThreadNotifyRoutine	Windows 2000	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetcreatethreadnotifyroutine
PsSetCreateThreadNotifyRoutineEx	Windows 10 (1507)	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetcreatethreadnotifyroutineex
PsSetLoadImageNotifyRoutine	Windows 2000	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetloadimagenotifyroutine
PsSetLoadImageNotifyRoutineEx	Windows 10 (1709)	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetloadimagenotifyroutineex
KeRegisterBugCheckCallback	Windows 2000	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-keregisterbugcheckcallback
KeRegisterBugCheckReasonCallback	Windows XP SP1	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-keregisterbugcheckreasoncallback
CmRegisterCallback	Windows XP	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-cmregistercallback
CmRegisterCallbackEx	Windows Vista	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-cmregistercallbackex

1 We assume Windows 7 minimum supported client here and below.

IoRegisterShutdownNotification	Windows 2000	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-ioregistershutdownnotification
IoRegisterLastChanceShutdownNotification	Windows 2000	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-ioregisterlastchanceshutdownnotification
ObRegisterCallbacks	Windows Vista SP1	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-obregistercallbacks
SeRegisterLogonSessionTerminatedRoutine	Windows NT 3.51 SP5	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-seregisterlogonsessionterminatedroutine
SeRegisterLogonSessionTerminatedRoutineEx	Windows 10	Undocumented
PoRegisterPowerSettingCallback	Windows Vista	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-poregisterpowersettingcallback
PoRegisterCoalescingCallback	Windows 8	Undocumented
DbgSetDebugPrintCallback	Windows Vista	Undocumented
IoRegisterFsRegistrationChange	Windows NT 3.51	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-ioregisterfsregistrationchange
DbgkLkmdRegisterCallback	Windows 7	Undocumented
PsRegisterPicoProvider	Windows 10 (1607)	Undocumented
KeRegisterNmiCallback	Windows 2003	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-keregisternmicallback
PsRegisterSiloMonitor	Windows 10 (1607)	https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-psregistersilomonitor
EmProviderRegister	Windows Vista	Undocumented

3. Implementation

3.1. Ps* Notify Routines

PsSetCreateProcessNotifyRoutine, PsSetCreateProcessNotifyRoutineEx, PsSetCreateProcessNotifyRoutineEx2, PsSetCreateThreadNotifyRoutine, PsSetCreateThreadNotifyRoutineEx, PsSetLoadImageNotifyRoutine, PsSetLoadImageNotifyRoutineEx.

These routines implementation represent EX_CALLBACK pointers array of fixed size (64 items starting from Windows 7) with not exported counter variable (one per each type – Process/Thread/Image). Before querying actual callback function from array pointer, array pointer must be decoded.

EX_CALLBACK structure defined as

```
typedef struct _EX_CALLBACK {
    EX_FAST_REF RoutineBlock;
} EX_CALLBACK, *PEX_CALLBACK;
```

EX_FAST_REF structure defined as

```
typedef struct _EX_FAST_REF {
    union {
        PVOID Object;
#if defined (_WIN64)
        ULONG_PTR RefCnt : 4;
#else
        ULONG_PTR RefCnt : 3;
#endif
        ULONG_PTR Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;
```

EX_FAST_REF RefCnt field here used to keep track of object references. Under x64 we are interested in remaining 60 bits, excluding RefCnt.

Decoded pointer will represent another structure called EX_CALLBACK_ROUTINE_BLOCK and defined as:

```
typedef struct _EX_CALLBACK_ROUTINE_BLOCK {
    EX_RUNDOWN_REF RundownProtect;
    PVOID Function; //PEX_CALLBACK_FUNCTION
    PVOID Context;
} EX_CALLBACK_ROUTINE_BLOCK, *PEX_CALLBACK_ROUTINE_BLOCK;
```

Example query pseudo code, PspRoutineArray is a kernel array variable that holds list of EX_CALLBACKS, it is different for each type — Process/Thread/Image:

```
EX_CALLBACKS Callbacks[MAX_CALLBACKS];

ReadMemory(PspRoutineArray, MAX_CALLBACKS * sizeof(EX_CALLBACK));

for (Index = 0; Index < MAX_CALLBACKS; Index++) {
      PVOID CallbackBlockRoutineAddress = (PVOID)(Callbacks[Index].RoutineBlock.Value & ~MAX_FAST_REFS)[2];

      EX_CALLBACK_ROUTINE_BLOCK CallbackBlockRoutine = ReadMemory(CallbackBlockRoutineAddress);
      PVOID CallbackFunction = CallbackBlockRoutine.Function;
}
```

Where CallbackFunction is actual pointer to driver routine responsible for handling this callback.

2 Where MAX_FAST_REFS is 15 for x64

3.2. KeBugCheck Notify Routines

KeRegisterBugCheckCallback, KeRegisterBugCheckReasonCallback.

Implemented as doubly linked lists whose heads are *KeBugCheckCallbackHead* and *KeBugCheckReasonCallbackHead* respectively. Each entry described by KBUGCHECK_CALLBACK_RECORD and KBUGCHECK_CALLBACK_REASON structures respectively.

KBUGCHECK_CALLBACK_RECORD

```
typedef struct _KBUGCHECK_CALLBACK_RECORD {
    LIST_ENTRY Entry;
    PVOID CallbackRoutine;
    PVOID Buffer;
    ULONG Length;
    PUCHAR Component;
    ULONG_PTR Checksum;
    UCHAR State;
} KBUGCHECK_CALLBACK_RECORD,
*PKBUGCHECK_CALLBACK_RECORD;
```

KBUGCHECK_CALLBACK_REASON

```
typedef struct _KBUGCHECK_REASON_CALLBACK_RECORD {
    LIST_ENTRY Entry;
    PVOID CallbackRoutine;
    PUCHAR Component;
    ULONG_PTR Checksum;
    KBUGCHECK_CALLBACK_REASON Reason;
    UCHAR State;
} KBUGCHECK_REASON_CALLBACK_RECORD,
*PKBUGCHECK_REASON_CALLBACK_RECORD;
```

CallbackRoutine is a pointer to driver defined handler routine. Enumerating these type of callback require doubly linked list walking.

3.3. Configuration Manager Callbacks

CmRegisterCallback, CmRegisterCallbackEx.

Implemented as doubly linked list whose head is *CallbackListHead*. Each entry described by CM_CALLBACK_CONTEXT_BLOCK and subject of changes between Windows versions. However there is a generic definition of this structure which is enough to enumerate registered callbacks.

```
typedef struct _CM_CALLBACK_CONTEXT_BLOCK {
    LIST_ENTRY CallbackListEntry;
    LONG PreCallListHead;
    LARGE_INTEGER Cookie;
    PVOID CallerContext;
    PVOID Function; //PEX_CALLBACK_FUNCTION
    UNICODE_STRING Altitude;
    LIST_ENTRY ObjectContextListHead;
} CM_CALLBACK_CONTEXT_BLOCK, *PCM_CALLBACK_CONTEXT_BLOCK;
```

Function is a pointer to driver defined handler routine. Enumerating this type of callback require doubly linked list walking.

3.4. Io Shutdown Notifications

IoRegisterShutdownNotification, IoRegisterLastChanceShutdownNotification.

Implemented as doubly linked lists whose heads are *IopNotifyShutdownQueueHead* and *IopNotifyLastChanceShutdownQueueHead* respectively. Each entry described by SHUTDOWN_PACKET structure.

```
typedef struct _SHUTDOWN_PACKET {
    LIST_ENTRY ListEntry;
    PDEVICE_OBJECT DeviceObject;
} SHUTDOWN_PACKET, *PSHUTDOWN_PACKET;
```

DeviceObject represent device created by driver. To query processing code we should read DeviceObject→DriverObject and look for IRP_MJ_SHUTDOWN in DRIVER_OBJECT MajorFunction array. Enumerating this type of callback require doubly linked list walking.

```
DRIVER_OBJECT LocalDriverObjectDump = ReadMemory(Entry.DeviceObject.DriverObject);
PVOID Routine = LocalDriverObjectDump.MajorFunction[IRP_MJ_SHUTDOWN];
```

3.5. Object Type Callbacks

ObRegisterCallbacks.

Implemented as doubly linked list whose head is an OBJECT_TYPE structure field *CallbackList*. OBJECT_TYPE is one of the key Ob Manager structures and subject of change between Windows versions. While generally structure looks the same, it part – another structure OBJECT_TYPE_INITIALIZER changes frequently and thus affecting parent structure size. There is a 4 variants of OBJECT_TYPE_INITIALIZER since 7 up to current Windows 11 21H2 (22000) at moment of writing this document. OBJECT_TYPE definition:

```
typedef struct _OBJECT_TYPE {
    LIST_ENTRY TypeList;
    UNICODE_STRING Name;
    PVOID DefaultObject;
    UCHAR Index;
    ULONG TotalNumberOfObjects;
    ULONG TotalNumberOfHandles;
    ULONG HighWaterNumberOfObjects;
    ULONG HighWaterNumberOfHandles;
    OBJECT_TYPE_INITIALIZER TypeInfo; //size may vary
    EX_PUSH_LOCK TypeLock;
    ULONG Key;
    LIST_ENTRY CallbackList;
} OBJECT_TYPE, POBJECT_TYPE;
```

CallbackList has **+0xC0** offset for Windows 7 (including SP1) and **+0xC8** for everything else up to Windows 11 21H2 (22000). Since this CallbackList is a part of OBJECT_TYPE it is by design part of all object types available in Windows. However this callback mechanisms only supported by *PsProcessType*, *PsThreadType* and *ExDesktopObjectType* since Windows 10. Internally Windows handles this with help of OBJECT_TYPE_INITIALIZER→ObjectTypeFlags.**SupportsObjectCallbacks** bit flag. This flag is only set for Process/Thread and Desktop on Windows 10.

Each callback entry in CallbackList represent the following structure:

```
typedef struct _OB_CALLBACK_CONTEXT_BLOCK {
    LIST_ENTRY CallbackListEntry;
    OB_OPERATION Operations;
    ULONG Flags;
    struct _OB_REGISTRATION* Registration;
    POBJECT_TYPE ObjectType;
    PVOID PreCallback; //POB_PRE_OPERATION_CALLBACK
    PVOID PostCallback; //POB_POST_OPERATION_CALLBACK
    EX_RUNDOWN_REF RundownReference;
} OB_CALLBACK_CONTEXT_BLOCK, *POB_CALLBACK_CONTEXT_BLOCK;
```

Most important fields are highlighted. Note that *Registration* field type is NOT compatible with *OB_CALLBACK_REGISTRATION* defined in wdm.h. It actual definition is private and declared below:

```
typedef struct _OB_REGISTRATION {
    USHORT Version;
    USHORT RegistrationCount;
    PVOID RegistrationContext;
    UNICODE_STRING Altitude;
    OB_CALLBACK_CONTEXT_BLOCK* CallbackContext;
} OB_REGISTRATION, *POB_REGISTRATION;
```

Enumeration of registered callbacks can be done by doubly linked list walking.

3.6. Session Notifications

SeRegisterLogonSessionTerminatedRoutine, SeRegisterLogonSessionTerminatedRoutineEx.

Implemented as single linked lists whose heads are *SeFileSystemNotifyRoutinesHead* and *SeFileSystemNotifyRoutinesHeadEx* respectively. Each entry described by the following structure:

```
typedef struct _SEP_LOGON_SESSION_TERMINATED_NOTIFICATION {
    struct _SEP_LOGON_SESSION_TERMINATED_NOTIFICATION *Next;
    PVOID CallbackRoutine; //PSE_LOGON_SESSION_TERMINATED_ROUTINE
} SEP_LOGON_SESSION_TERMINATED_NOTIFICATION, *PSEP_LOGON_SESSION_TERMINATED_NOTIFICATION;
```

Note that for SeRegisterLogonSessionTerminatedRoutineEx structure is slightly different, extended in it tail, however required fields are on the same offset so another definition is not required. Enumeration of registered callbacks can be done by single linked list walking.

3.7. Power Settings Callbacks

PoRegisterPowerSettingCallback.

Implemented as doubly linked list with head *PopRegisteredPowerSettingCallbacks*.

Each entry described by POP_POWER_SETTING_REGISTRATION structure which is a subject of change between Windows version.

Before Windows 10 1607 (RS1 14393)

```
typedef struct _POP_POWER_SETTING_REGISTRATION_V1 {
    LIST_ENTRY Link;
    ULONG Tag;
    PVOID CallbackThread;
    UCHAR UnregisterOnReturn;
    UCHAR UnregisterPending;
    GUID Guid;
    PVOID LastValue;
    PVOID Callback;
    PVOID Context;
    PDEVICE_OBJECT DeviceObject;
} POP_POWER_SETTING_REGISTRATION_V1,
*PPOP_POWER_SETTING_REGISTRATION_V1;
```

After Windows 10 1607 (RS1 14393)

```
typedef struct _POP_POWER_SETTING_REGISTRATION_V2 {
    LIST_ENTRY Link;
    ULONG Tag;
    PVOID CallbackThread;
    UCHAR UnregisterOnReturn;
    UCHAR UnregisterPending;
    GUID Guid;
    GUID Guid2;
    PVOID LastValue;
    PVOID Callback;
    PVOID Context;
    PDEVICE_OBJECT DeviceObject;
} POP_POWER_SETTING_REGISTRATION_V2,
*PPOP_POWER_SETTING_REGISTRATION_V2;
```

Note that tail of V2 is incorrect for newest Windows 10 versions, however this does not affect enumeration which can be done by doubly linked list walking.

3.8. DebugPrint Callbacks

DbgSetDebugPrintCallback.

For unknown reason this API is not documented however used by software since Vista release, for example by MS SysInternals DbgView. Implemented as doubly linked list whose head is *RtlpDebugPrintCallbackList*. Each entry described by the following structure:

```
typedef struct _RTL_CALLBACK_REGISTER {
    ULONG Flags;
    EX_RUNDOWN_REF RundownReference;
    PVOID DebugPrintCallback;
    LIST_ENTRY ListEntry;
} RTL_CALLBACK_REGISTER, *PRTL_CALLBACK_REGISTER;
```

Since ListEntry is a tail of this structure actual address of entry must be calculated before query/read memory. For example:

```
RTL_CALLBACK_REGISTER *Next = ListEntry.Flink - FIELD_OFFSET(RTL_CALLBACK_REGISTER, ListEntry);
ReadMemory(Next);
ListEntry.Flink = Next.ListEntry.Flink;
```

Enumeration is doubly linked list walking with next entry calculation as above.

3.9. Coalescing Callbacks

PoRegisterCoalescingCallback.

Undocumented callback used by Windows Cache Manager, Configuration Manager and NTFS driver. It was introduced in Windows 8. Before Windows 10 RS4 (17134) implemented as an array of callbacks *PopCoalescingCallbackRoutine* with maximum eight (8) elements. **Warning**: Windows 10 RS3 (16299) has increased capacitity to thirty-two (32) elements. Since Windows 10 RS4 (17763) it was updated to be unlimited capacity linked list *PopCoalRegistrationList*. Each entry is described by the following structure:

Windows 8 → Windows 10 1709 (RS3 16299)

```
typedef struct _PO_COALESCING_CALLBACK_V1 {
    EX_PUSH_LOCK PushLock;
    PVOID CoalescingCallback;
    PVOID SelfPtr;
    PPO_COALESCING_CALLBACK Callback;
    BOOLEAN ClientOrServer;
    PVOID Context;
} PO_COALESCING_CALLBACK_V1, *
PPO_COALESCING_CALLBACK_V1;
```

After Windows 10 1803 (RS4 17134)

```
typedef struct _PO_COALESCING_CALLBACK_V2 {
    EX_PUSH_LOCK PushLock;
    PVOID CoalescingCallback;
    PVOID SelfPtr;
    PPO_COALESCING_CALLBACK Callback;
    BOOLEAN ClientOrServer;
    PVOID Context;
    LIST_ENTRY Link;
    EX_CALLBACK ExCallback;
} PO_COALESCING_CALLBACK_V2, *
PPO_COALESCING_CALLBACK_V2;
```

Since V2 Link field is a tail of this structure actual address of entry must be calculated before query/read memory. Enumeration for V1 is similar to Ps* routines and for V2 it is a double linked list walking with next entry calculation as below:

```
PO_COALESCING_CALLBACK_V2* Next = ListEntry.Flink - FIELD_OFFSET(PO_COALESCING_CALLBACK_V2, Link);
ReadMemory(Next);
ListEntry.Flink = Next.ListEntry.Flink;
```

3.10. IoFs Change Notifications

IoRegisterFsRegistrationChange.

Implemented as doubly linked list whose head is *IopFsNotifyChangeQueueHead*. Each entry described by the following structure:

```
typedef struct _NOTIFICATION_PACKET {
    LIST_ENTRY ListEntry;
    PDRIVER_OBJECT DriverObject;
    PVOID NotificationRoutine; //PDRIVER_FS_NOTIFICATION
} NOTIFICATION_PACKET, *PNOTIFICATION_PACKET;
```

Enumeration is doubly linked list walking.

3.11. DbgK Callbacks

DbgkLkmdRegisterCallback.

Undocumented callback mechanism used by NtSystemDebugControl service. Implemented as fixed size array of EX_CALLBACK structures. Before querying actual callback function from array pointer, array pointer must be decoded, decoding is the same as in case of Ps*Notify routines. Array can contain contain maximum eight (8) elements. Currently used by win32k.

3.12. Pico Provider Routines

PsRegisterPicoProvider.

Undocumented callback mechanism working as a part of Windows Subsystem for Linux (WSL). Implemented as fixed size structure which members are subject of change between Windows versions. The first element is a size of structure in bytes. This structure holds pointers to callbacks and data filled by LXCORE driver during ***LxInitialize*** call.

3.13. Nonmaskable Interrupt Callbacks

KeRegisterNmiCallback.

Implemented as single linked list whose head is *KiNmiCallbackListHead*. Each entry described by the following structure:

```
typedef struct _KNMI_HANDLER_CALLBACK {
    struct _KNMI_HANDLER_CALLBACK* Next;
    PNMI_CALLBACK Callback;
    PVOID Context;
    PVOID Handle;
} KNMI_HANDLER_CALLBACK, *PKNMI_HANDLER_CALLBACK;
```

Enumeration of registered callbacks can be done by single linked list walking.

3.14. Silo Monitor Callbacks

PsRegisterSiloMonitor.

Implemented as double linked list whose head is *PspSiloMonitorList*. Each entry described by the following structure:

```
typedef struct _SERVER_SILO_MONITOR {
    LIST_ENTRY ListEntry;
    UCHAR Version;
    BOOLEAN MonitorHost;
    BOOLEAN MonitorExistingSilos;
    UCHAR Reserved[5];
    SILO_MONITOR_CREATE_CALLBACK CreateCallback;
    SILO_MONITOR_TERMINATE_CALLBACK TerminateCallback;
    union {
        PUNICODE_STRING DriverObjectName;
        PUNICODE_STRING ComponentName;
    };
} SERVER_SILO_MONITOR, * PSERVER_SILO_MONITOR;
```

Enumeration is doubly linked list walking.

3.15. Errata Manager Callbacks

EmProviderRegister.

Implemented as double linked list whose head is *EmpCallbackListHead*. Each entry described by the following structure:

```
typedef struct _EMP_CALLBACK_DB_RECORD {
    GUID CallbackId;
    PVOID CallbackFunc;
    LONG_PTR CallbackFuncReference;
    PVOID Context;
    SINGLE_LIST_ENTRY List;
    SINGLE_LIST_ENTRY CallbackDependencyListHead;
    ULONG NumberOfStrings;
    ULONG NumberOfNumerics;
    ULONG NumberOfEntries;
    struct _EMP_ENTRY_DB_RECORD* EntryList[1];
} EMP_CALLBACK_DB_RECORD, * PEMP_CALLBACK_DB_RECORD;
```

Since List is in a middle of this structure actual address of entry must be calculated before query/read memory. For example:

```
PEMP_CALLBACK_DB_RECORD *Next = ListEntry.Flink – FIELD_OFFSET(EMP_CALLBACK_DB_RECORD, List);
ReadMemory(Next);
ListEntry.Flink = Next.ListEntry.Flink;
```

Enumeration is doubly linked list walking with next entry calculation as above.

4.0. Search Patterns

4.1. PspCreateProcessNotifyRoutine

```
PsSetCreateProcessNotifyRoutine = GPA(mappedNtoskrnl, “PsSetCreateProcessNotifyRoutine”);
PspSetCreateProcessNotifyRoutine = SearchForCallOrJmp(PsSetCreateProcessNotifyRoutine);
Index = 0; Rel = 0;
ptrCode = PspSetCreateProcessNotifyRoutine;
do {
      if (!disasm(I, ptrCode + Index))
            break;
      if (I.Length == 7)
      If (ptrCode[Index] == 0x4C && ptrCode[Index + 1] == 0x8D)
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
      Index += I.Length;
} while (Index < 128);
if (Rel) PspCreateProcessNotifyRoutine = ConvertAddressWithBase(ptrCode, ntoskrnlBase, Rel, mappedNtoskrnl);
```

4.2. PspCreateThreadNotifyRoutine

```
ptrCode = GPA(mappedNtoskrnl, "PsRemoveCreateThreadNotifyRoutine");
Index = 0; Rel = 0;
do {
      if (!disasm(I, ptrCode + Index))
            break;
      if (I.Length == 7)
      If ((ptrCode[Index] == 0x48 || ptrCode[Index] == 0x4C) && ptrCode[Index + 1] == 0x8D)
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
      Index += I.Length;
} while (Index < 128);
if (Rel) PspCreateThreadNotifyRoutine = ConvertAddressWithBase(ptrCode, ntoskrnlBase, Rel, mappedNtoskrnl);
```

4.3. PspLoadImageNotifyRoutine

```
ptrCode = GPA(mappedNtoskrnl, "PsRemoveLoadImageNotifyRoutine");
...
      if (I.Length == 7)
      If ((ptrCode[Index] == 0x48 || ptrCode[Index] == 0x4C) && ptrCode[Index + 1] == 0x8D)
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
```

4.4. KeBugCheckCallbackHead

```
ptrCode = GPA(mappedNtoskrnl, “KeRegisterBugCheckCallback”);
...
do {
...
      if (I.Length == 7)
      If ((ptrCode[Index] == 0x48 || ptrCode[Index] == 0x4C) &&
            (ptrCode[Index + 1] == 0x8D) && (ptrCode[Index + I.Length] == 0x48))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 512);
...
```

4.5. KeBugCheckReasonCallbackHead

```
ptrCode = GPA(mappedNtoskrnl, "KeRegisterBugCheckReasonCallback");
...
do {
...
      if (I.Length == 7)
            if (((ptrCode[Index] == 0x48) || (ptrCode[Index] == 0x4C)) &&
            (ptrCode[Index + 1] == 0x8D) &&
            ((ptrCode[Index + hs.len] == 0x48) || (ptrCode[Index + hs.len] == 0x83)))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 512);
...
```

4.6. IopNotifyShutdownQueueHead

```
ptrCode = GPA(mappedNtoskrnl, "IoRegisterShutdownNotification");
...
do {
...
      if (I.Length == 7)
      if (((ptrCode[Index] == 0x48) || (ptrCode[Index] == 0x4C)) &&
            (ptrCode[Index + 1] == 0x8D))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 128);
...
```

4.7. IopNotifyLastChanceShutdownQueueHead

```
ptrCode = GPA(mappedNtoskrnl, "IoRegisterLastChanceShutdownNotification");
...
do {
...
      if (I.Length == 7)
      if (((ptrCode[Index] == 0x48) || (ptrCode[Index] == 0x4C)) &&
            (ptrCode[Index + 1] == 0x8D))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 128);
...
```

4.8. CallbackListHead (Configuration Manager)

```
ptrCode = GPA(mappedNtoskrnl, "CmUnRegisterCallback");
...
do {
...
      if (I.Length == 5)
      if ((ptrCode[Index] == 0x48) && (ptrCode[Index + 1] == 0x8D) && (ptrCode[Index + 2] == 0x54))
      {
            if (!disasm(I_next, ptrCode + Index + I.Length))
                  break;
            if (I_next.Length == 7) {
                  if ((ptrCode[Index + I.Length] == 0x48) &&
                        (ptrCode[Index + I.Length + 1] == 0x8D) &&
                        (ptrCode[Index + I.Length + 2] == 0x0D))
                        {
                              ptrCodeOffset = Index + I.Length + I_next.Length;
                              Rel = *(PLONG)(ptrCode + Index + I.Length + 3);
                        }
            }
      }
...
} while (Index < 256);
...
```

4.9. CallbackList (Object Type)

```
ObjectRefAddr = ObReferenceObjectAddr(ObjectType);
OBJECT_TYPE_V = SelectObjectTypeVersion(NtBuildNumber);
CallbackList = ObjectRefAddr + FIELD_OFFSET(OBJECT_TYPE_V, CallbackList);
```

where ObReferenceObjectAddr is either locating address of kernel mode object of required type (Process/Thread/Desktop) or parsing Ob directory.

4.10. SeFileSystemNotifyRoutinesHead

```
ptrCode = GPA(mappedNtoskrnl, "SeRegisterLogonSessionTerminatedRoutine");
...
do {
...
      if (I.Length == 7)
            if ((ptrCode[Index] == 0x48) &&
                (ptrCode[Index + 1] == 0x8B) && (ptrCode[Index + 2] == 0x05))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 128);
...
```

4.11. SeFileSystemNotifyRoutinesExHead

```
ptrCode = GPA(mappedNtoskrnl, "SeRegisterLogonSessionTerminatedRoutineEx");
...
do {
...
      if (I.Length == 7)
            if ((ptrCode[Index] == 0x48) &&
                (ptrCode[Index + 1] == 0x8B) && (ptrCode[Index + 2] == 0x05))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 128);
...
```

4.12. PopRegisteredPowerSettingCallbacks

```
ptrCode = GPA(mappedNtoskrnl, “PoRegisterPowerSettingCallback”);
...
do {
...
      if (I.Length == 7)
            if ((ptrCode[Index] == 0x48) &&
                 (ptrCode[Index + 1] == 0x8D) &&
                 (ptrCode[Index + 2] == 0x0D) && (ptrCode[Index + 7] == 0x48))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 512);
...
```

4.13. CoalescingCallbacks

```
ptrCode = GPA(mappedNtoskrnl, "PoRegisterCoalescingCallback");
checkByte = (IsWinVer < Win10RS4) ? 0x0D : 0x15;
...
do {
…
      if (I.Length == 7)
            if ((ptrCode[Index] == 0x48) &&
                  (ptrCode[Index + 1] == 0x8D) &&
                  (ptrCode[Index + 2] == checkByte)
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 256);
```

4.14. RtlpDebugPrintCallbackList

```
//First, search for DbgpInsertDebugPrintCallback
ptrCode = GPA(mappedNtoskrnl, "DbgSetDebugPrintCallback");
...
do {
...
      if (I.Length == 5) { //jmp or call
            if ((ptrCode[Index] == 0xE9) ||
                (ptrCode[Index] == 0xE8))
            {
                Rel = *(PLONG)(ptrCode + Index + 1);
                break;
            }
        }
        if (hs.len == 6) { //jz
            if (ptrCode[Index] == 0x0F) {
                Rel = *(PLONG)(ptrCode + Index + 2);
                break;
            }
}
...
} while (Index < 64);
...
if (Rel) ptrCode = ptrCode + Index + (I.Length) + Rel;
else break;
```

```
//Next search for RtlpDebugPrintCallbackList in DbgpInsertDebugPrintCallback

do {
...
      if (I.Length == 7)
             if ((ptrCode[Index] == 0x48) &&
                 (ptrCode[Index + 1] == 0x8D) &&
                 ((ptrCode[Index + 2] == 0x15) || (ptrCode[Index + 2] == 0x0D)) &&
                   (ptrCode[Index + hs.len] == 0x48))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 512);
...
```

4.15. IopFsNotifyChangeQueueHead

```
ptrCode = GPA(mappedNtoskrnl, "IoUnregisterFsRegistrationChange");
...
do {
...
      if (I.Length == 7)
            if ((ptrCode[Index] == 0x48) &&
                (ptrCode[Index + 1] == 0x8D) &&
                (ptrCode[Index + 2] == 0x05) &&
                (ptrCode[Index + 7] == 0xEB))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 512);
...
```

4.16. DbgkLkmdCallbacks

```
ptrCode = GPA(mappedNtoskrnl, “DbgkLkmdUnregisterCallback”);
...
do {
...
      if (I.Length == 7)
      if (((ptrCode[Index] == 0x4C) || (ptrCode[Index] == 0x48)) &&
            (ptrCode[Index + 1] == 0x8D))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
...
} while (Index < 64);
```

4.17. PspPicoProviderRoutines

```
ptrCode = GPA(mappedNtoskrnl, “PsRegisterPicoProvider”);
...
do {
...
      if (I.Length == 7)
      if ((ptrCode[Index] == 0x0F) &&
          (ptrCode[Index + 1] == 0x11) &&
          (ptrCode[Index + 2] == 0x05))
      {
            Rel = *(PLONG)(ptrCode + Index + 3);
            break;
      }
…
} while (Index < 256);
```

4.18. KiNmiCallbackListHead

```
If (WinVer < Win10TH1) break; //does not support anything below Win10.
ptrCode = GPA(mappedNtoskrnl, "KeDeregisterNmiCallback");
…
if (WinVer < Win10RS3) {
      c = 0;
      do {
            if (I.Length == 7) {
                  if (ptrCode[Index] == 0x48 &&
                        ptrCode[Index + 1] == 0x8D &&
                        ptrCode[Index + 2] == 0x0D)
                    {
                        c += 1;
                    }
                }

                if (c > 2) {
                    Rel = *(PLONG)(ptrCode + Index + 3);
                    break;
                }

      } while (Index < 256);

} else
{
      Rel = 0;
      do {
            if (I.Length == 5) {
                  //
                  // Find call to KiDeregisterNmiSxCallback
                  //
                  if (ptrCode[Index] == 0xE8) {
                        Rel = *(PLONG)(ptrCode + Index + 1);
                        break;
                  }
            }

      } while (Index < 64);
```

```
	if (Rel != 0) {
		ptrCode = ptrCode + Index + I.Length + Rel;
		Index = 0
		Rel = 0
		c = 0
		//Scan KiDeregisterNmiSxCallback
		do {
			if (I.Length == 7)
				if (ptrCode[Index] == 0x48 &&
					ptrCode[Index + 1] == 0x8D &&
					ptrCode[Index + 2] == 0x0D)
				{
					c += 1;
				}
			}
			if (c > 1) {
				Rel = *(PLONG)(ptrCode + Index + 3);
				break;
			}
	}

}
```

4.19. PspSiloMonitorList

```
ptrCode = GPA(mappedNtoskrnl, “PsStartSiloMonitor”);
…

      do {
            if (I.Length == 7) {
                  if (ptrCode[Index] == 0x48 &&
                        ptrCode[Index + 1] == 0x8D &&
                        ptrCode[Index + 2] == 0x0D &&
                        ptrCode[Index + I.Length) == 0x48)
                  {
                        Rel = *(PLONG)(ptrCode + Index + 3);
                        break;
                  }
            }
      } while (Index < 512);
```

4.20. EmpCallbackListHead

```
// Find EmpSearchCallbackDatabase first

PAGE_Section = GetSectionByName("PAGE", &SectionSize);
BYTE g_EmpSearchCallbackDatabase[] = { 0x48, 0x8B, 0x4E, 0xF8, 0x48, 0x85, 0xC9 };
BYTE g_EmpSearchCallbackDatabase2[] = { 0x49, 0x8B, 0x4A, 0xF8, 0x48, 0x85, 0xC9 };


If (WinVer < Win81) {

      signature = EmpSearchCallbackDatabase;

} else {

      signature = EmpSearchCallbackDatabase2;

}

…
ptrCode = FindPattern(PAGE_Section, SectionSize, signature, sizeof(signature));
Index += sizeof(signature);
Rel = 0;

do {
// Find EmpSearchCallbackDatabase call
            if (I.Length == 5) {
                  if (ptrCode[Index] == 0xE8) {
                        Rel = *(PLONG)(ptrCode + Index + 1);
                        break;
                  }
            }
...
        } while (Index < 64);
```

```
if (Rel != 0) {
      ptrCode = ptrCode + Index + I.Length + Rel;
      Index = 0;
      Rel = 0;
      do {
…
            if (I.Length == 7) {
                  if (ptrCode[Index] == 0x48) {
                        Rel = *(PLONG)(ptrCode + Index + 3);
                        break;
                  }
            }
      } while (Index < 32);
}
```

5.0. Callback Object Type

Callback is a kernel mode object type. Driver call *ExCreateCallback* to create new object with “Callback” type and register driver specified routine with *ExRegisterCallback*. Callback objects are usually located in dedicated \Callback object directory. This object type described by the following structure:

Windows 7

```
typedef struct _CALLBACK_OBJECT {
    ULONG Signature;
    KSPIN_LOCK Lock;
    LIST_ENTRY RegisteredCallbacks;
    BOOLEAN AllowMultipleCallbacks;
    UCHAR reserved[3];
} CALLBACK_OBJECT, *PCALLBACK_OBJECT;
```

Windows 8.1 and above

```
typedef struct _CALLBACK_OBJECT_V2 {
    ULONG Signature;
    KSPIN_LOCK Lock;
    LIST_ENTRY RegisteredCallbacks;
    BOOLEAN AllowMultipleCallbacks;
    LIST_ENTRY ExpCallbackList;
} CALLBACK_OBJECT_V2, * PCALLBACK_OBJECT_V2;
```

Where list entries are described by CALLBACK_REGISTRATION structure:

```
typedef struct _CALLBACK_REGISTRATION {
    LIST_ENTRY Link;
    PCALLBACK_OBJECT CallbackObject;
    PVOID CallbackFunction; //PCALLBACK_FUNCTION
    PVOID CallbackContext;
    ULONG Busy;
    BOOLEAN UnregisterWaiting;
} CALLBACK_REGISTRATION, *PCALLBACK_REGISTRATION;
```

By walking *RegisteredCallback* doubly linked list for given object of type “Callback” we can enumerate registered callback routines.

Examples of callback objects:

\Callback\ProcessorAdd – dynamically track changes in the processor population;

\Callback\SeImageVerificationDriverInfo – when callback object registered with *SeRegisterImageVerificationCallback* it will be invoked each time a driver image is loaded in memory;

\Callback\PowerState – Invoked when the system switches from AC to DC power or vice versa, the system power policy changes as the result of a user or application request, a transition to a system sleep or shutdown state is imminent.

6.0. Code Integrity Callbacks

Managed by CI.DLL and implemented as fixed size array of pointers which is initialized by CI.DLL during the call of

ntoskrnl.exe!SepInitializeCodeIntegrity → CI.DLL!CiInitialize → CI.DLL!CipInitialize.

Provides ntoskrnl with interface access to CI.DLL implemented methods. Declared by ntoskrnl as **g_CiCallbacks** (Windows 7) and **SeCiCallbacks** (Windows 8 and above). Since Windows 10 this data is protected by PatchGuard and contents of the interface are different for desktop Windows and Xbox (for Xbox it is initialized with own *Xci** functions provided through api-set).

Since Windows 8 the data structure holding these callbacks has been redesigned from simple array to a structure with additional first element added. This element is size of structure in bytes.

Since Windows 10 RS1 (14393) another additional field has been prepended to the end of the structure. It holds *NTDDI_value* which you can usually find in sdkddkver.h official MS header file. This value is unique for each Windows 10/11 version.

Extraction of this data structure is variadic because of different implementations for Windows versions. In general for Windows 7 up to Windows 10 RS4 it is pattern search inside ntoskrnl to locate ***SepInitializeCodeIntegrity*** where ***CiInitialize*** called with parameters where one of it is pointer to callbacks data.

Since Windows 10 RS5 ntoskrnl exports new service table function called ***NtCompareSigningLevels***. Inside this function implemented as direct call of ***SeCiCallbacks*** array pointer by offset. Which give us opportunity to query and calculate actual address of ***SeCiCallbacks*** without doing much of Windows-version specific pattern scanning.

Note that contents of ***SeCiCallbacks*** for Windows 10/11 can be a subject of change during Windows cumulative update process. For example, there has been at least one case when new methods have been added to the callbacks table by regular patch without Windows version major upgrade.

7.0. Copyrights and References

Document Version 3.6

References

http://geoffchappell.com/studies/windows/km/ntoskrnl/api/index.htm?tx=23
http://redplait.blogspot.com/
http://eretik.omegahg.com/index.htm
https://github.com/swwwolf/wdbgark