Enumerating Kernel Notification Callback Routines (x64)

This document applies to:

- Windows 7
- Windows 7 SP1
- Windows 8
- Windows 8.1
- Windows 10 (TH1 up to 22H2)
- Windows 11 (21H2 up to 25H2)

Note: This document assumes Windows 7 as the minimum supported client version.

Table of Contents

1. Introduction

List of common kernel callback routines, their implementation, and enumeration methods.

Important note: Since all operations here involve kernel-mode components, a kernel-mode driver is required for memory read operations.

The examples below use **kldbgdrv** (WinDbg Kernel Local Debugging Driver). This driver provides stable memory read functionality via the ***KdSystemDebugControl*** routine in ***ntoskrnl.exe*** (requires Debug Mode enabled and ***SeDebugPrivilege*** assigned). Any other custom driver can be used instead, such as *rkhdrv50* or *wodbgdrv*.

Overview of Callbacks
Callbacks (often called "notification routines" or "notifiers") are used by driver developers to receive notifications when specific events occur. They are distinct from Callback objects, a separate kernel-mode object type that also facilitates notifications.

Common callbacks fall into two categories:

- **Notification-Only Callbacks:** Used solely to monitor events.
- **Behavior-Modifying Callbacks:** Can alter event outcomes.

Due to their power, callbacks are frequently abused by malicious actors, fraudulent software, or bloatware.

Enumeration Challenges
There is no official method to enumerate these callbacks. Their implementation is undocumented and subject to change across Windows versions. Proper enumeration also requires interacting with locks and synchronization mechanisms, which:

- Are not exported as functions or variables.
- Cannot be accessed or invoked from user mode.

Attempting to read this data from user mode risks retrieving incoherent results due to lack of synchronization.

2. Callbacks Table

Routine Name	Minimum supported client	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
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
IoRegisterPlugPlayNotification	Windows 2000	https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioregisterplugplaynotification

3. Implementation

3.1. Ps* Notify Routines

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

These routines use an `EX_CALLBACK` **pointers array** of fixed size (64 items starting from Windows 7) with a non-exported counter variable (one per type: **Process, Thread, Image**). Before querying the actual callback function from the array pointer, the pointer must be decoded.

EX_CALLBACK structure definition

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

EX_FAST_REF structure definition

```
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;
```

The `EX_FAST_REF` structure's *RefCnt* **field** (reference count) is used to track object references. On x64 systems, the **remaining 60 bits** (excluding RefCnt) store the encoded pointer.

The decoded pointer points to an `EX_CALLBACK_ROUTINE_BLOCK` structure, 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 an array holding `EX_CALLBACKS` entries (unique for Process/Thread/Image types):

```
#define MAX_CALLBACKS 64 // Fixed size since Windows 7

#ifdef _WIN64
#define MAX_FAST_REFS 15
#else
#define MAX_FAST_REFS 7
#endif

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);

      EX_CALLBACK_ROUTINE_BLOCK CallbackBlockRoutine = ReadMemory(CallbackBlockRoutineAddress);
      PVOID CallbackFunction = CallbackBlockRoutine.Function; // Driver's callback handler
}
```

3.2. KeBugCheck Notify Routines

KeRegisterBugCheckCallback, KeRegisterBugCheckReasonCallback.

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

KBUGCHECK_CALLBACK_RECORD structure definition

```
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 structure definition

```
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 this type of callback requires traversing the doubly-linked list.

3.3. Configuration Manager Callbacks

CmRegisterCallback, CmRegisterCallbackEx.

Implemented as doubly-linked list whose head is *CallbackListHead*. Each entry is described by `CM_CALLBACK_CONTEXT_BLOCK` structure, which is subject to changes between Windows versions. However, a generic definition of this structure exists that is sufficient 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 requires traversing the doubly-linked list.

3.4. Io Shutdown Notifications

IoRegisterShutdownNotification, IoRegisterLastChanceShutdownNotification.

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

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

DeviceObject represents 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 requires traversing the doubly-linked list

```
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 Object Manager structures and subject to changes between Windows versions. While generally structure looks the same, it part – another structure `OBJECT_TYPE_INITIALIZER` changes frequently and thus affecting parent structure size. As of the time of writing this document, there are four distinct variants of the `OBJECT_TYPE_INITIALIZER` structure across Windows versions, spanning from Windows 7 to Windows 11 25H2 (build 27842). Below is a generic definition of `OBJECT_TYPE` structure:

```
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 field has and offset **+0xC0** in Windows 7 (including SP1) and **+0xC8** in all later versions up to Windows 11 25H2 (build 27842). Since this *CallbackList* is a part of `OBJECT_TYPE` it is by design part of all object types available in Windows. However, callback mechanisms are **only supported** for *PsProcessType* (Process objects), *PsThreadType* (Thread objects) and *ExDesktopObjectType* (Desktop Objects) starting with Windows 10. Windows controls callback support via the **SupportsObjectCallbacks** bit flag in the `OBJECT_TYPE_INITIALIZER`→*ObjectTypeFlags* field. **As of Windows 10/11**, this flag is enabled **only** for Process, Thread, and Desktop object types.

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 it reverse-engineered definition shown below:

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

Enumerating this type of callback requires traversing the doubly-linked list.

3.6. Session Notifications

SeRegisterLogonSessionTerminatedRoutine, SeRegisterLogonSessionTerminatedRoutineEx.

Implemented as singly-linked lists whose heads are *SeFileSystemNotifyRoutinesHead* and *SeFileSystemNotifyRoutinesHeadEx* respectively. Each entry is 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*, the structure is slightly different – it is extended in its tail, however required fields are at the same offsets so another definition is not required. Enumerating this type of callback requires traversing the singly-linked list.

3.7. Power Settings Callbacks

PoRegisterPowerSettingCallback.

Implemented as doubly-linked list with head *PopRegisteredPowerSettingCallbacks*.

Each entry is described by `POP_POWER_SETTING_REGISTRATION` structure which is a subject to changes between Windows versions.

Before Windows 10 1607 (RS1, build 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, build 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 10/11 versions. However, enumerating this type of callback still can be done by traversing the doubly-linked list.

3.8. DebugPrint Callbacks

DbgSetDebugPrintCallback.

For unknown reasons, this API is undocumented yet has been used by software since Vista's release (for example, by Microsoft Sysinternals DbgView). Implemented as a doubly-linked list whose head is *RtlpDebugPrintCallbackList*. Each entry is 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 at the tail of this structure, the actual address of the entry must be calculated before querying/reading memory. For example:

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

Enumerating this type of callback requires traversing the doubly-linked list with next entry calculation as shown 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 (build 17134), implemented as an array of callbacks *PopCoalescingCallbackRoutine* with maximum eight (8) elements.

Warning: Windows 10 RS3 (build 16299) increased capacity to thirty-two (32) elements. Since Windows 10 RS4 (build 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 build 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 build 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 at the tail of this structure, the actual address of entry must be calculated before querying/reading memory. Enumeration for V1 is similar to Ps* routines, and for V2 it is a doubly-linked list traversing 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 is described by the following structure:

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

Enumerating this type of callback requires traversing the doubly-linked list.

3.11. DbgK Callbacks

DbgkLkmdRegisterCallback.

This **undocumented** callback mechanism is tied to *NtSystemDebugControl* service service and implemented as a **fixed-size array** of `EX_CALLBACK` structures. Before accessing the callback function pointers, **the array pointer must be decoded**, similar to the obfuscation used in the Ps*Notify routines. The array holds **up to 8 elements**, each representing a callback. Primarily leveraged by *win32k.sys* (the kernel-mode Win32 subsystem driver) for internal event handling.

3.12. Pico Provider Routines

PsRegisterPicoProvider.

This **undocumented** callback mechanism operates as part of the **Windows Subsystem for Linux (WSL)**. It is implemented as a **fixed-size structure,** though its members are **subject to changes between Windows versions**. The **first element** is the size of the structure in bytes. This structure contains **pointers to callback routines** and data populated by the *LXCORE.sys* driver during the ***LxInitialize*** call. This structure's layout (offsets and member definitions) **varies across Windows versions**, requiring version-specific handling. Extraction is **not straightforward** and relies on pattern-matching of reverse-engineering the *LXCORE.sys* driver.

3.13. Nonmaskable Interrupt Callbacks

KeRegisterNmiCallback.

Implemented as a singly-linked list whose head is *KiNmiCallbackListHead*. Each entry is 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;
```

Enumerating this type of callback requires traversing the singly-linked list.

3.14. Silo Monitor Callbacks

PsRegisterSiloMonitor.

Implemented as a doubly-linked list whose head is *PspSiloMonitorList*. Each entry is 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;
```

Enumerating this type of callback requires traversing the doubly-linked list.

3.15. Errata Manager Callbacks

EmProviderRegister.

Implemented as a doubly-linked list whose head is *EmpCallbackListHead*. Each entry is 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 operations. For example:

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

Enumerating this type of callback requires traversing the doubly-linked list with next entry calculation as shown above.

3.16. Plug and Play (PnP) Notification Callbacks

IoRegisterPlugPlayNotification.

Implementation depends on type of notification requested and is different on old Windows versions. In NT6 and above kernel it is implemented as a doubly-linked list whose head is *PnpDeviceClassNotifyList*. Each entry is described by the following structure:

```
typedef struct _DEVICE_CLASS_NOTIFY_ENTRY {
    LIST_ENTRY ListEntry;
    IO_NOTIFICATION_EVENT_CATEGORY EventCategory;
    ULONG SessionId;
    HANDLE SessionHandle;
    PDRIVER_NOTIFICATION_CALLBACK_ROUTINE CallbackRoutine;
    PVOID Context;
    PDRIVER_OBJECT DriverObject;
    USHORT RefCount;
    BOOLEAN Unregistered;
    PKGUARDED_MUTEX Lock;
    PERESOURCE EntryLock;
    GUID ClassGuid;
} DEVICE_CLASS_NOTIFY_ENTRY, * PDEVICE_CLASS_NOTIFY_ENTRY;
```

Enumerating this type of callback requires traversing the doubly-linked list.

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

where *ObReferenceObjectAddress* either:

- Locates the address of a kernel-mode object of the required type (Process/Thread/Desktop), or
- Parses Object Manager directory to query that address

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 };
BYTE g_EmpSearchCallbackDatabase3[] = { 0x4B, 0x8B, 0x0C, 0xDC, 0x48, 0x85, 0xC9, 0x74, 0x48 };

If (WinVer < Win81) {

      signature = EmpSearchCallbackDatabase;

} else if (WinVer <= Win11_23H2) {

      signature = EmpSearchCallbackDatabase2;

} else {
      signature = EmpSearchCallbackDatabase3;
}

…
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);
}
```

4.21. PnpDeviceClassNotifyList

```
ptrCode = GPA(mappedNtoskrnl, “IoRegisterPlugPlayNotification”);
...
BYTE g_PnpDeviceClassNotifyList_SubPattern_7601[] = { 0xF7, 0xE1 };
BYTE g_PnpDeviceClassNotifyList_SubPattern_9200[] = { 0xC1, 0xEA, 0x02, 0x6B, 0xD2, 0x0D };
BYTE g_PnpDeviceClassNofityList_SubPattern_9600_26080[] = { 0xC1, 0xEA, 0x02, 0x6B, 0xC2, 0x0D
};
...
switch (WinVer) {
case Win7:

      signature = g_PnpDeviceClassNotifyList_SubPattern_7601;
       break

case WinVer80:

      signature = g_PnpDeviceClassNotifyList_SubPattern_9200;
      break;

default:
      signature = g_PnpDeviceClassNofityList_SubPattern_9600_26080;
      break;

}
...
```

```
ptrCode = FindPattern(ptrCode, 1024, signature, sizeof(signature));
Index += sizeof(signature);
Rel = 0;

do {

            if ((I.Length == 7) &&
                (I.Flags & F_PREFIX_REX) &&
                (I.Flags & F_DISP32) &&
                (I.Flags & F_MODRM) &&
                (I.opcode == 0x8D))
            {
                Rel = *(PLONG)(ptrCode + Index + 3);
                break;
            }

            Index += hs.len;

} while (Index < 64);
```

5.0. Callback Object Type

Callback is a kernel-mode object type. Driver call *ExCreateCallback* to create a new object of “Callback” type and registers a driver-specified routine with *ExRegisterCallback*. Callback objects are typically located in the dedicated \Callback object directory. Below is a definition of structure describing the Callback object type:

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*, this system is implemented as a **fixed-size array of pointers**, initialized during the execution of:

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

The array provides *ntoskrnl.exe* with an interface to methods implemented in *CI.DLL*. It is declared in *ntoskrnl.exe* as:

- *g_CiCallbacks* (Windows 7)
- *SeCiCallbacks* (Windows 8 and later).

Key Changes Across Windows Versions:

1. **Windows 8 Redesign:**
   - The data structure changed from a simple array to a structure with an **added first element** (the size of the structure in bytes).
2. **Windows 10 and Later:**
   - Protected by **PatchGuard**.
   - The interface contents differ between **desktop Windows** and **Xbox** (Xbox uses *Xci** functions via an API set).
3. **Windows 10 RS1 (14393):**
   - A new **NTDDI version field** (from *sdkddkver.h*) was added to the end of the structure as marker. This value is unique to each Windows 10/11 version.

Extraction Methods:

- **Windows 7 to Windows 10 RS4:**

  Use pattern scanning within *ntoskrnl.exe* to locate *SepInitializeCodeIntegrity*, where *CiInitialize* is called with parameters pointing to the callbacks data.

- **Windows 10 RS5 and Later:**

  The *NtCompareSigningLevels* function (exported by *ntoskrnl.exe*) directly references the *SeCiCallbacks* array via a fixed offset, simplifying retrieval without version-specific pattern scans.

Important Notes:

- The contents of *SeCiCallback*s in Windows 10/11 may change during cumulative updates, even without major OS version upgrades. For example, Microsoft has added new methods to the table via routine patches.

7.0. Copyrights and References

Document Version 3.8

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