Lord Of The Ring0 - Part 2 | A tale of routines, IOCTLs and IRPs

idov31.github.io/2022/08/04/lord-of-the-ring0-p2.html

August 4, 2022

Prologue

In the last blog post, we had an introduction to kernel development and what are the difficulties when trying to load a driver and how to bypass it. In this blog, I will write more about callbacks, how to start writing a rootkit and the difficulties I encountered during my development of Nidhogg.

As I promised to bring both defensive and offensive points of view, we will create a driver that can be used for both blue and red teams - A process protector driver.

P.S: The name Nidhogg was chosen after the nordic dragon that lies underneath Yggdrasil :).

Talking with the user mode 101

A driver should be (most of the time) controllable from the user mode by some process, an example would be Sysmon - When you change the configuration, turn it off or on it tells its kernel part to stop performing certain operations, works by an updated policy or just shut down it when you decide to unload Sysmon. As kernel drivers, we have two ways to communicate with the user mode: Via DIRECT_IO or IOCTLs.The advantage of DIRECT_IO is that it is more simple to use and you have more control and the advantage of using IOCTLs is that it is safer and developer friendly. In this blog series, we will use the IOCTLs approach.

To understand what is an IOCTL better, let’s look at an IOCTL structure:

```
#define MY_IOCTL CTL_CODE(DeviceType, FunctionNumber, Method, Access)
```

The device type indicates what is the type of the device (different types of hardware and software drivers), it doesn’t matter much for software drivers will be the number but the convention is to use 0x8000 for 3rd software drivers like ours.

The second parameter indicates the function “index” in our driver, it could be any number but the convention suggests starting from 0x800.

The method parameter indicates how the input and output should be handled by the driver, it could be either METHOD_BUFFERED or METHOD_IN_DIRECT or METHOD_OUT_DIRECT or METHOD_NEITHER.

The last parameter indicates if the driver accepts the operation (FILE_WRITE_ACCESS) or the driver operates (FILE_READ_ACCESS) or the driver accepts and performs the operation (FILE_ANY_ACCESS).

To use IOCTLs, on the driver's initialization you will need to set a function that will parse an IRP and knows how to handle the IOCTLs, such a function is defined as followed:

```
NTSTATUS MyDeviceControl(
    [in] PDEVICE_OBJECT DeviceObject,
    [in] PIRP           Irp
);
```

IRP in a nutshell is a structure that represents an I/O request packet. You can read more about it in MSDN.

When communicating with the user mode we need to define two more things: The device object and the symbolic link. The device object is the object that handles the I/O requests and allows us as a user-mode program to communicate with the kernel driver. The symbolic link creates a linkage in the GLOBAL?? directory so the DeviceObject will be accessible from the user mode and usually looks like \??\DriverName.

Callback Routines

To understand how to use callback routines let's understand WHAT are they. The callback routine is a feature that allows kernel drivers to register for certain events, an example would be process operation (such as: getting a handle to process) and affect their result. When a kernel driver registers for an operation, it notifies "I'm interested in the certain event and would like to be notified whenever this event occurs" and then for each time this event occurs the driver is get notified and a function is executed.

One of the most notable ways to register for an operation is with the ObRegisterCallbacks function:

```
NTSTATUS ObRegisterCallbacks(
  [in]  POB_CALLBACK_REGISTRATION CallbackRegistration,
  [out] PVOID                     *RegistrationHandle
);

typedef struct _OB_CALLBACK_REGISTRATION {
  USHORT                    Version;
  USHORT                    OperationRegistrationCount;
  UNICODE_STRING            Altitude;
  PVOID                     RegistrationContext;
  OB_OPERATION_REGISTRATION *OperationRegistration;
} OB_CALLBACK_REGISTRATION, *POB_CALLBACK_REGISTRATION;

typedef struct _OB_OPERATION_REGISTRATION {
  POBJECT_TYPE                *ObjectType;
  OB_OPERATION                Operations;
  POB_PRE_OPERATION_CALLBACK  PreOperation;
  POB_POST_OPERATION_CALLBACK PostOperation;
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

Using this callback we can register for two types of OperationRegistration: ObjectPreCallback and ObjectPostCallback. The pre-callback happens before the operation is executed and the post-operation happens after the operation is executed and before the user gets back the output.

Using ObRegisterCallback you can register for this ObjectTypes of operations (You can see the full list defined in WDM.h):

- PsProcessType
- PsThreadType
- ExDesktopObjectType
- IoFileObjectType
- CmKeyObjectType
- ExEventObjectType
- SeTokenObjectType
- …

To use this function, you will need to create a function with a unique signature as follows (depending on your needs and if you are using PreOperation or PostOperation):

```
OB_PREOP_CALLBACK_STATUS PobPreOperationCallback(
  [in] PVOID RegistrationContext,
  [in] POB_PRE_OPERATION_INFORMATION OperationInformation
)

void PobPostOperationCallback(
  [in] PVOID RegistrationContext,
  [in] POB_POST_OPERATION_INFORMATION OperationInformation
)
```

Now that we understand better what callbacks are we can write our first driver - A kernel driver that protects a process.

Let’s build - Process Protector

To build a process protector we need to first understand how will it work. What we want is basic protection against any process that attempts to kill our process, the protected process could be our malicious program or our precious Sysmon agent. To perform the killing of a process the process that performs the killing will need a handle with the PROCESS_TERMINATE permissions, and before we said that we could register for certain events like a request for the handle to process. So as a driver, you could remove permissions from a handle and return a handle without specific permission which is in our case the PROCESS_TERMINATE permission.

To start with the development we will need a DriverEntry function:

```
#include <ntddk.h>

// Definitions
#define IOCTL_PROTECT_PID    CTL_CODE(0x8000, 0x800, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define PROCESS_TERMINATE 1

// Prototypes
DRIVER_UNLOAD ProtectorUnload;
DRIVER_DISPATCH ProtectorCreateClose, ProtectorDeviceControl;

OB_PREOP_CALLBACK_STATUS PreOpenProcessOperation(PVOID RegistrationContext,
POB_PRE_OPERATION_INFORMATION Info);

// Globals
PVOID regHandle;
ULONG protectedPid;

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    NTSTATUS status = STATUS_SUCCESS;
    UNICODE_STRING deviceName = RTL_CONSTANT_STRING(L"\\Device\\Protector");
    UNICODE_STRING symName = RTL_CONSTANT_STRING(L"\\??\\Protector");
    PDEVICE_OBJECT DeviceObject = nullptr;

    OB_OPERATION_REGISTRATION operations[] = {
        {
            PsProcessType,
            OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICATE,
            PreOpenProcessOperation, nullptr
        }
    };

    OB_CALLBACK_REGISTRATION reg = {
        OB_FLT_REGISTRATION_VERSION,
        1,
        RTL_CONSTANT_STRING(L"12345.6879"),
        nullptr,
        operations
    };

    ...
```

Before we continue let’s explain what’s going on, we defined a deviceName with our driver name (Protector) and a symbolic link with the same name (the symName parameter). We also defined an array of operations that we want to register for - In our case it is just the PsProcessType for each handle creation or handle duplication.

We used this array to finish the registration definition - the number 1 stands for only 1 operation to be registered, and the 12345.6879 defines the altitude. An altitude is a unique double number (but using a UNICODE_STRING to represent it) that is used to identify registration and relate it to a certain driver.

As you probably noticed, the DriverEntry is “missing” the RegistryPath parameter, to not write UNREFERENCED_PARAMETER(RegistryPath) we can just not write it and it will be unreferenced.

Now, let’s do the actual registration and finish the DriverEntry function:

```
...

    status = IoCreateDevice(DriverObject, 0, &deviceName, FILE_DEVICE_UNKNOWN, 0,
FALSE, &DeviceObject);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create device object (status=%08X)\n",
status));
        return status;
    }

    status = IoCreateSymbolicLink(&symName, &deviceName);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create symbolic link (status=%08X)\n",
status));
        IoDeleteDevice(DeviceObject);
        return status;
    }

    status = ObRegisterCallbacks(&reg, &regHandle);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to register the callback (status=%08X)\n",
status));
        IoDeleteSymbolicLink(&symName);
        IoDeleteDevice(DeviceObject);
        return status;
    }

    DriverObject->DriverUnload = ProtectorUnload;
    DriverObject->MajorFunction[IRP_MJ_CREATE] = DriverObject-
>MajorFunction[IRP_MJ_CLOSE] = ProtectorCreateClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ProtectorDeviceControl;

    KdPrint(("DriverEntry completed successfully\n"));
    return status;
}
```

Using the functions IoCreateDevice and IoCreateSymbolicLink we created a device object and a symbolic link. After we know our driver can be reached from the user mode we registered our callback with ObRegisterCallbacks and defined important major functions such as ProtectorCreateClose (will explain it soon) and ProtectorDeviceControl to handle the IOCTL.

The ProtectorUnload function is very simple and just does the cleanup like we did if the status wasn't successful: The next thing on the list is to implement the ProtectorCreateClose function. The function is responsible on complete the IRP, since in this driver we don't have multiple device objects and we are not doing much with it we can handle the completion of the relevant IRP in our DeviceControl function and for any other IRP just close it always with a successful status.

```
NTSTATUS ProtectorCreateClose(PDEVICE_OBJECT, PIRP Irp) {
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

The device control is also fairly simple as we have only one IOCTL to handle:

```
NTSTATUS ProtectorDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    NTSTATUS status = STATUS_SUCCESS;
    auto stack = IoGetCurrentIrpStackLocation(Irp);

    switch (stack->Parameters.DeviceIoControl.IoControlCode) {
        case IOCTL_PROTECT_PID:
        {
            auto size = stack->Parameters.DeviceIoControl.InputBufferLength;

            if (size % sizeof(ULONG) != 0) {
                status = STATUS_INVALID_BUFFER_SIZE;
                break;
            }

            auto data = (ULONG*)Irp->AssociatedIrp.SystemBuffer;
            protectedPid = *data;
            break;
        }
        default:
            status = STATUS_INVALID_DEVICE_REQUEST;
            break;
    }

    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

As you noticed, to see the IOCTL, get the input and for more operations in the future, we need to use the IRP's stack. I won't go over its entire structure but you can view it in MSDN. To make it clearer, when using the METHOD_BUFFERED option the input and output buffers are delivered via the SystemBuffer that is located within the IRP's stack.

After we got the stack and verified the IOCTL, we need to check our input because wrong input handling can cause a BSOD. When the input verification is completed all we have to do is just change the protectedPid to the wanted PID.

With the DeviceControl and the CreateClose functions, we can create the last function in the kernel driver - The PreOpenProcessOperation.

```
OB_PREOP_CALLBACK_STATUS PreOpenProcessOperation(PVOID, POB_PRE_OPERATION_INFORMATION
Info) {
    if (Info->KernelHandle)
        return OB_PREOP_SUCCESS;

    auto process = (PEPROCESS)Info->Object;
    auto pid = HandleToULong(PsGetProcessId(process));

    // Protecting our pid and removing PROCESS_TERMINATE.
    if (pid == protectedPid) {
        Info->Parameters->CreateHandleInformation.DesiredAccess &=
~PROCESS_TERMINATE;
    }

    return OB_PREOP_SUCCESS;
}
```

Very simple isn't it? Just logic and the opposite value of the PROCESS_TERMINATE and we are done.

Now, we have left only one thing to make sure and it is to allow our driver to register for operation registration, **it can be done within the project settings in Visual Studio in the linker command line and just add /integritycheck switch**.

After we finished with the kernel driver part let's go to the user-mode part.

Protector's User mode Part

The user-mode part is even simple as we just need to create a handle for the device object and send the wanted PID.

```
#include <iostream>
#include <Windows.h>

int main(int argc, const char* argv[]) {
    DWORD bytes;

    if (argc != 1) {
        std::cout << "Usage: " << argv[0] << " <pid>" << std::endl;
        return 1;
    }

    DWORD pid = atoi(argv[1]);
    HANDLE device = CreateFile(L"\\\\.\\Protector", GENERIC_READ | GENERIC_WRITE, 0,
nullptr, OPEN_EXISTING, 0, nullptr);

    if (device == INVALID_HANDLE_VALUE) {
        std::cout << "Failed to open device" << std::endl;
        return 1;
    }

    success = DeviceIoControl(device, IOCTL_PROTECT_PID, &pid, sizeof(pid), nullptr,
0, &bytes, nullptr);
    CloseHandle(device);

    if (!success) {
        std::cout << "Failed in DeviceIoControl: " << GetLastError() << std::endl;
        return 1;
    }

    std::cout << "Protected process with pid: " << pid << std::endl;
    return 0;
}
```

Congratulations on writing your very first functional kernel driver!

Bonus - Anti-dumping

To prevent a process from being dumped all we have to do is just remove more permissions such as PROCESS_VM_READ, PROCESS_DUP_HANDLE and PROCESS_VM_OPERATION. An example can be found in Nidhogg's ProcessUtils file.

Conclusion

In this blog, we got a better understanding of how to write a driver, how to communicate it and how to use callbacks. In the next blog, we will dive more into this world and learn more new things about kernel development.

I hope that you enjoyed the blog and I'm available on Twitter, Telegram and by Mail to hear what you think about it! This blog series is following my learning curve of kernel mode development and if you like this blog post you can check out Nidhogg on GitHub.