Lord Of The Ring0 - Part 4 | The call back home

idov31.github.io/2023/02/24/lord-of-the-ring0-p4.html

February 24, 2023

Prologue

In the last blog post, we learned some debugging concepts, understood what is IOCTL how to handle it and started to learn how to validate the data that we get from the user mode - data that cannot be trusted and a handling mistake can cause a blue screen of death.

In this blog post, I'll explain the different types of callbacks and we will write another driver to protect registry keys.

Kernel Callbacks

We started to talk about this subject in the 2nd part, so if you haven't read it yet read it here and come back as this blog is based on the knowledge you have learned in the previous ones.

For starters, let's see what type of callbacks we're going to learn about today:

- Pre / Post operations (can be registered with `ObRegisterCallbacks` and talked about it in the 2nd part).
- PsSet*NotifyRoutine.
- CmRegisterCallbackEx.

Each of the mentioned callbacks has its purpose and difference and the most important thing to know is to get the right tool for the job, so for each type, I will also give an example of how it can be used in different scenarios.

ObRegisterCallbacks

ObRegisterCallbacks is a function that allows you to register a callback of your choice for certain events (process, thread, and much more) before or after they're happening. To register a callback you need to give the following structure:

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

`Version` **MUST** be OB_FLT_REGISTRATION_VERSION.

`OperationRegistrationCount` is the number of registered callbacks.

`Altitude` is a unique identifier in form of a string with this pattern `#define OB_CALLBACKS_ALTITUDE L"XXXXX.XXXX"` where `X` is a number. It is mandatory to define one so the OS will be able to identify your driver and determine the load order if you don't define it or if the `Altitude` isn't unique the registration will fail.

`RegistrationContext` is the handle that will be used later on to Unregister the callbacks.

Finally, `OperationRegistration` is an array that contains all of your registered callbacks. `OperationRegistration` and every callback have this structure:

```
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;
```

`ObjectType` is the type of operation that you want to register to. Some of the most common types are `*PsProcessType` and `*PsThreadType`. It is worth mentioning that although you can enable more types (like IoFileObjectType) this will trigger PatchGuard and cause your computer to BSOD, so unless PatchGuard is disabled it is highly not recommended to enable more types. If you still want to enable more types, you can do so by using this like so:

```
typedef struct _OBJECT_TYPE
{
 struct _LIST_ENTRY TypeList;
 struct _UNICODE_STRING Name;
 VOID* DefaultObject;
 UCHAR Index;
 ULONG TotalNumberOfObjects;
 ULONG TotalNumberOfHandles;
 ULONG HighWaterNumberOfObjects;
 ULONG HighWaterNumberOfHandles;
 struct _OBJECT_TYPE_INITIALIZER_TEMP TypeInfo;
 struct _EX_PUSH_LOCK_TEMP TypeLock;
 ULONG Key;
 struct _LIST_ENTRY CallbackList;
} OBJECT_TYPE, * POBJECT_TYPE;

POBJECT_TYPE_TEMP ObjectTypeTemp = (POBJECT_TYPE_TEMP)*IoFileObjectType;
ObjectTypeTemp->TypeInfo.SupportsObjectCallbacks = 1;
```

`Operations` are the kind of operations that you are interested in, it can be `OB_OPERATION_HANDLE_CREATE` and/or `OB_OPERATION_HANDLE_DUPLICATE` for a handle creation or duplication.

`PreOperation` is an operation that will be called before the handle is opened and `PostOperation` will be called after it is opened. In both cases, you are getting important information through `OB_PRE_OPERATION_INFORMATION` or `OB_POST_OPERATION_INFORMATION` such as a handle to the object, the type of the object the return status, and what type of operation (`OB_OPERATION_HANDLE_CREATE` or `OB_OPERATION_HANDLE_DUPLICATE`) occurred. Both of them must **ALWAYS** return `OB_PREOP_SUCCESS` , if you want to change the return status, you can change the `ReturnStatus` that you got from the operation information, but do not return anything else.

After you registered this kind of callback, you can remove certain permissions from the handle (for example: If you don't want to allow a process to be closed, you can just remove the `PROCESS_TERMINATE` permission as we did in part 2 of the series) or manipulate the object itself (if it is a process, you can change the EPROCESS structure).

As you can see, these kinds of operations are very useful for both rootkits and AVs/EDRs to protect their user mode component. Usually, if you have a user mode part you will want to use some of these callbacks to make sure your process/thread is protected properly and cannot be killed easily.

PsSet*NotifyRoutine

Unlike `ObRegisterCallbacks` PsSet notifies routines are not responsible for a handle opening or duplicating operation but for monitoring creation/killing and loading operations, while the most notorious ones are `PsSetCreateProcessNotifyRoutine` , `PsSetCreateThreadNotifyRoutine` and `PsSetLoadImageNotifyRoutine` all of them are heavily used by AVs/EDRs to monitor for certain process/thread creations and DLL loading. Let's break it down, and talk about each function separately and what you can do with it.

`PsSetCreateProcessNotifyRoutine` receives a function of type `PCREATE_PROCESS_NOTIFY_ROUTINE` which looks like so:

```
void PcreateProcessNotifyRoutine(
  [in] HANDLE ParentId,
  [in] HANDLE ProcessId,
  [in] BOOLEAN Create
)
```

`ParentId` is the PID of the process that attempts to create or kill the target process. `ProcessId` is the PID of the target process. `Create` indicates whether it is a create or kill operation.

The most common example of using this kind of routine is to watch certain processes and if there is an attempt to create a forbidden process (e.g. create a cmd directly under Winlogon), you can kill it. Another example can be of creating a "watchdog" for a certain process and if it is killed by an unauthorized process, restart it.

`PsSetCreateThreadNotifyRoutine` receives a function of type `PCREATE_THREAD_NOTIFY_ROUTINE` which looks like so:

```
void PcreateThreadNotifyRoutine(
  [in] HANDLE ProcessId,
  [in] HANDLE ThreadId,
  [in] BOOLEAN Create
)
```

`ProcessId` is the PID of the process. `ThreadId` is the TID of the target thread. `Create` indicates whether it is a create or kill operation.

A simple example of using this kind of routine is if an EDR injected its library into a process, make sure that the library's thread is getting killed.

`PsSetLoadImageNotifyRoutine` receives a function of type `PLOAD_IMAGE_NOTIFY_ROUTINE` which looks like so:

```
void PloadImageNotifyRoutine(
  [in, optional] PUNICODE_STRING FullImageName,
  [in]           HANDLE ProcessId,
  [in]           PIMAGE_INFO ImageInfo
)
```

`FullImageName` is the name of the loaded image (a note here: it is not only DLLs and can be also EXE for example). `ProcessId` is the PID of the target process. `ImageInfo` is the most interesting part and contains a struct of type `IMAGE_INFO`:

```
typedef struct _IMAGE_INFO {
  union {
    ULONG Properties;
    struct {
      ULONG ImageAddressingMode : 8;
      ULONG SystemModeImage : 1;
      ULONG ImageMappedToAllPids : 1;
      ULONG ExtendedInfoPresent : 1;
      ULONG MachineTypeMismatch : 1;
      ULONG ImageSignatureLevel : 4;
      ULONG ImageSignatureType : 3;
      ULONG ImagePartialMap : 1;
      ULONG Reserved : 12;
    };
  };
  PVOID  ImageBase;
  ULONG  ImageSelector;
  SIZE_T ImageSize;
  ULONG  ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

The most important properties in my opinion are `ImageBase` and `ImageSize` , using these you can inspect and analyze the image pretty efficiently. A simple example is if an attacker injects a DLL into LSASS, an EDR can inspect the image and unload it if it finds it malicious. If the `ExtendedInfoPresent` option is available, it means that this struct is of type `IMAGE_INFO_EX` :

```
typedef struct _IMAGE_INFO_EX {
  SIZE_T              Size;
  IMAGE_INFO          ImageInfo;
  struct _FILE_OBJECT *FileObject;
} IMAGE_INFO_EX, *PIMAGE_INFO_EX;
```

As you can see, here you also get the `FILE_OBJECT` which is a handle for the file that is backed on the disk. With that information, you can also check for reflective DLL injection (a loaded DLL without any file backed on the disk) and it opens a door for you to monitor for more injection methods that don't have a file on the disk.

These kinds of functions are usually used more for EDRs and AVs rather than rootkits, because as you can see it provides insights that are more useful for monitoring rather than doing malicious operations but that doesn't mean it doesn't have a use at all. For example, a rootkit can use the `PsSetLoadImageNotifyRoutine` to make sure that no AV/EDR agent is injected into it.

CmRegisterCallbackEx

CmRegisterCallbackEx is responsible to register a registry callback that can monitor and interfere with various registry operations such as registry key creation, deletion, querying and more. Like the `ObRegisterCallbacks` functions, it receives a unique altitude and the callback function. Let's focus on the Registry callback function:

```
NTSTATUS ExCallbackFunction(
  [in]           PVOID CallbackContext,
  [in, optional] PVOID Argument1,
  [in, optional] PVOID Argument2
)
```

`CallbackContext` is the context that was passed on the function registration with `CmRegisterCallbackEx` . `Argument1` is a variable that contains the information of what operation was made (e.g. deletion, creation, setting value) and whether it is a post-operation or pre-operation. `Argument2` is the information itself that is delivered and its type matches the class that was specified in `Argument1` .

Using this callback, a rootkit can do many operations, from blocking a change to a specific registry key, denying setting a specific value or hiding registry keys and values.

An example is a rootkit that saves its configuration in the registry and then hides it using this callback. To give another practical example, we will create now another driver - a driver that can protect registry keys from deletion.

Registry Protector

First, let’s start with the `DriverEntry`:

```
#define DRIVER_PREFIX "MyDriver: "
#define DRIVER_DEVICE_NAME L"\\Device\\MyDriver"
#define DRIVER_SYMBOLIC_LINK L"\\??\\MyDriver"
#define REG_CALLBACK_ALTITUDE L"31102.0003"

PVOID g_RegCookie;

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);
    NTSTATUS status = STATUS_SUCCESS;

    UNICODE_STRING deviceName = RTL_CONSTANT_STRING(DRIVER_DEVICE_NAME);
    UNICODE_STRING symbolicLink = RTL_CONSTANT_STRING(DRIVER_SYMBOLIC_LINK);
    UNICODE_STRING regAltitude = RTL_CONSTANT_STRING(REG_CALLBACK_ALTITUDE);

    // Creating device and symbolic link.
    status = IoCreateDevice(DriverObject, 0, &deviceName, FILE_DEVICE_UNKNOWN, 0,
FALSE, &DeviceObject);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed to create device: (0x%08X)\n", status));
        return status;
    }

    status = IoCreateSymbolicLink(&symbolicLink, &deviceName);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed to create symbolic link: (0x%08X)\n",
status));
        IoDeleteDevice(DeviceObject);
        return status;
    }

    // Registering the registry callback.
    status = CmRegisterCallbackEx(RegNotify, &regAltitude, DriverObject, nullptr,
&g_RegContext, nullptr);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed to register registry callback: (0x%08X)\n",
status));
        IoDeleteSymbolicLink(&symbolicLink);
        IoDeleteDevice(DeviceObject);
        return status;
    }

    DriverObject->DriverUnload = MyUnload;
    return status;
}
```

We added to the standard `DriverEntry` initializations (Creating DeviceObject and symbolic link) `CmRegisterCallbackEx` to register our `RegNotify` callback. Note that we saved the `g_RegContext` as a global variable, as it will be used soon in the `MyUnload` function to unregister the driver when the `DriverUnload` is called.

```
void MyUnload(PDRIVER_OBJECT DriverObject) {
    KdPrint((DRIVER_PREFIX "Unloading...\n"));
    NTSTATUS status = CmUnRegisterCallback(g_RegContext);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed to unregister registry callbacks: (0x%08X)\n",
status));
    }

    UNICODE_STRING symbolicLink = RTL_CONSTANT_STRING(DRIVER_SYMBOLIC_LINK);
    IoDeleteSymbolicLink(&symbolicLink);
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

In `MyUnload` , we didn’t just unload the driver but also made sure to unregister our callback using the `g_RegContext` from before.

```
NTSTATUS RegNotify(PVOID context, PVOID Argument1, PVOID Argument2) {
    PCUNICODE_STRING regPath;
    UNREFERENCED_PARAMETER(context);
    NTSTATUS status = STATUS_SUCCESS;

    switch ((REG_NOTIFY_CLASS)(ULONG_PTR)Argument1) {
        case RegNtPreDeleteKey: {
            REG_DELETE_KEY_INFORMATION* info =
static_cast<REG_DELETE_KEY_INFORMATION*>(Argument2);

            // To avoid BSOD.
            if (!info->Object)
                break;

            status = CmCallbackGetKeyObjectIDEx(&g_RegContext, info->Object, nullptr,
&regPath, 0);

            if (!NT_SUCCESS(status))
                break;

            if (!regPath->Buffer || regPath->Length < 50)
                break;

            if (_wcsnicmp(LR"(SYSTEM\CurrentControlSet\Services\MaliciousService)",
regPath->Buffer, 50) == 0) {
                KdPrint((DRIVER_PREFIX "Protected the malicious service!\n"));
                status = STATUS_ACCESS_DENIED;
            }

            CmCallbackReleaseKeyObjectIDEx(regPath);
        }
        break;
    }

    return status;
}
```

Let’s break down what we’ve done here. First, we checked what is the type of operation and chose to respond only for `RegNtPreDeleteKey`. When we know that `Argument2` contains information of type REG_DELETE_KEY_INFORMATION we can cast to it.

After the cast, we can use the `Object` parameter to access the registry key itself to get the key’s path. To do that, we can use `CmCallbackGetKeyObjectIDEx`:

```
NTSTATUS CmCallbackGetKeyObjectIDEx(
  [in]            PLARGE_INTEGER   Cookie,
  [in]            PVOID            Object,
  [out, optional] PULONG_PTR       ObjectID,
  [out, optional] PCUNICODE_STRING *ObjectName,
  [in]            ULONG            Flags
);
```

`Cookie` is our global `g_RegContext` variable. `Object` is the registry key object. `ObjectID` is a unique registry identifier for our needs it can be null. `*ObjectName` is the output registry key path, make sure it is in the kernel format. `Flags` must be 0.

When you got the `ObjectName` it is just a matter of comparing it and the key that you want to protect and if it matches you can change the status to `STATUS_ACCESS_DENIED` to block the operation.

You can see a full implementation of the different registry operations handling in Nidhogg's Registry Utils.

Conclusion

In this blog, we learned about the different types of kernel callbacks and created our registry protector driver. In the next blog, we will learn what kernel R/W primitives mean and how we can use that to execute code, deepen our knowledge of interacting with the user mode and write a simple driver that can perform AMSI bypass to apply this knowledge.

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.