

x64 平台 SSDT HOOK 之 NtResumeThread 注入

作者: ExploitCN

1、前言

本文的思路、代码、调试参考了大量资料，历经艰难的调试后，现在可直接用 VS2019 在 win7_sp1_x64 系统上运行。

编写本文的目的，重点在于解释代码编写的思路，做到举一反三的目的，为软件逆向、代码 HOOK 打开思路，提升逆向代码编写水平。

2、前提知识

2.1 SSDT 表

System Service Descript Table(SSDT) :主要处理 Kernel32.dll 中的系统调用，如 openProcess,ReadFile 等，主要在 ntoskrnl.exe 中实现（微软有给出 ntoskrnl 源代码）

因为 x64 位中 ssdt 表是加密的，ssdt 中的每一项占 4 个字节但并不是对应的系统服务的地址，因为 x64 中地址为 64 位而 ssdt 每一项只有 4 个字节 32 位所以无法直接存放服务的地址。其实际存储的 4 个字节的前 28 位表示的是对应的系统服务相对于 SSDT 表基地址的偏移，而后 4 位如果对应的服务的参数个数小于 4 则其值为 0，不小于 4 则为参数个数减去 4。所以我们在 ssdt hook 时向 ssdt 表项中填入的函数得在 ntoskrnl.exe 模块中，原因是因为函数到 SSDT 表基地址的偏移大小小于 4 个字节。所以我们选取一个 ntoskrnl.exe 中很少使用的函数 KeBugCheckEx 作为中转函数，将需要 hook 的 ssdt 项的改为 KeBugCheckEx 函数，然后在 inlinehook KeBugCheck 函数，jmp 到我们的函数中进行过滤。

2.2 MSR 寄存器

MSR (Model Specific Register) 是 x86 架构中的概念，指的是在 x86 架构处理器中，一系列用于控制 CPU 运行、功能开关、调试、跟踪程序执行、监测 CPU 性能等方面的寄存器。

MSR 寄存器的雏形开始于 Intel 80386 和 80486 处理器，到 Intel Pentium 处理器的时候，Intel 就正式引入 RDMSR 和 WRMSR 两个指令用于读和写 MSR 寄存器，这个时候 MSR 就算被正式引入。在引入 RDMSR 和 WRMSR 指令的同时，也引入了

CPUID 指令，该指令用于指明具体的 CPU 芯片中，哪些功能是可用的，或者这些功能对应的 MSR 寄存器是否存在，软件可以通过 CPUID 指令查询某些功能是否在当前 CPU 上是否支持。

去 AMD 的官网，下载 AMD 芯片手册：

<https://developer.amd.com/resources/developer-guides-manuals/>

之后，查找 MSRC000_0082,如下图：

MSRC000_0082 [Long Mode SYSCALL Target Address] (Core::X86::Msr::STAR64)	
Read-write. Reset: 0000_0000_0000_0000h.	
_ccd[1:0]_lthree0_core[7:0]_thread[1:0]; MSRC000_0082	
Bits	Description
63:0	LSTAR: long mode target address. Read-write. Reset: 0000_0000_0000_0000h. Target address for 64-bit mode calling programs. The address stored in this register must be in canonical form (if not canonical, a #GP fault occurs).

可见，C000_0082 是 SYSCALL_Target_Address，在 windbg 里面，我们看看这个地址是什么。

```
1 5: kd> rdmsr c00000082
2 msr[c00000082] = fffff800`0408eec0
3 5: kd> uf fffff800`0408eec0
4 Flow analysis was incomplete, some code may be missing
5 nt!KiSystemCall64:
6 fffff800`0408eec0 0f01f8          swapgs
7 fffff800`0408eec3 654889242510000000 mov     qword ptr gs:[10h],rsp
8 fffff800`0408eecb 65488b2425a8010000 mov     rsp,qword ptr gs:[1A8h]
9 . . . . .
1 fffff800`0408eff2 4c8d1547a92300 lea     r10,[nt!KeServiceDescriptorTable
0 (fffff800`042c9940)]
1 fffff800`0408eff9 4c8d1d80a92300 lea     r11,[nt!KeServiceDescriptorTableShadow
1 (fffff800`042c9980)]
1 fffff800`0408f000 f7830001000080000000 test dword ptr [rbx+100h],80h
2
```

KeServiceDescriptorTableShadow 就是 SSDT 表的地址。记住这里，后面代码中要用到。

2.3 跳转指令

x64 下面常用的跳转指令：

1、mov rax,addr;jmp rax

48 B8 qword_addr -> mov rax,qword_addr

FF E0 -> jmp rax

2、push / ret

68 dword_addr -> push dword_addr

c3 -> ret

注意：push 的立即数是 dword，但由于对齐的关系，实际占用 64 位。如果跳转地址超过 2GB，要使用其他指令。push 的立即数不能是 64 位。

3、jmp、call

FF15 dword_offset qword_addr -> call[qword_addr]

FF25 dword_offset qword_addr -> jmp [qword_addr]

简单说明下：

在下条指令偏移 dword_offset 处，是要跳转的地址 qword_addr。当 dword_offset=0x00000000 时，qword_addr 就在指令的第 6 个字节。

但是在 x86 下，：

FF15 dword_addr -> call [dword_addr]

FF25 dword_addr -> jmp [dword_addr]

没有偏移，后面直接跟着绝对地址。

2.4 shellcode 执行出现 movaps 错误

我在写 shellcode 的时候，出现了 movaps 错误，一直以为代码有问题，后来经过排查、搜索，发现问题是由于在 shellcode 里面 call 函数之前，要保证 rsp 是 16 字节对齐的。我在这里耽误了一些时间的，你在使用的时候，尤其要注意。

2.5 为什么要自己实现 GetModuleHandle 和 GetProcAddress

首先要明白，win7_sp1_x64 下，CreateProcess 的创建流程是这样的：

```
1 Kernel32!CreateProcess
2 Kernel32!CreateProcessW
3 Kernel32!CreateProcessInternalW
4 ntdll!NtCreateProcessEx
5 ntdll!NtCreateThread
6 ntdll!NtResumeThread
```

此时，主进程的大部分创建工作已经完成（尤其是 ntdll.dll 已经加载，尤其重要），主线程刚刚被创建，此时还没有加载除 ntdll.dll 之外的其他 dll 而 GetModuleHandle 和 GetProcAddress 函数在 kernel32.dll 里，所以需要自己去实现这两个函数，获取

dll 的句柄和函数地址。kernel32.dll 是在进程创建完成、主线程初始化输入表时才载入的。

2.6 实现 GetModuleHandle 和 GetProcAddress 概略思路

2.6.1 GetModuleHandle 实现

GetModuleHandle 是为了获取 dll 基址，现在要获取 ntdll.dll 基址，是通过暴力搜索内存实现的。

主要用到的两个函数是：

ZwQueryVirtualMemory

ZwQuerySystemInformation

其中，

ZwQuerySystemInformation 用到的类型是 SystemEmulationBasicInformation。很重要，我找了很多资料，才找到这个参数。

ZwQueryVirtualMemory 用到的类型是 MemoryBasicInformation，和另外一个未文档化的参数 MEMORY_SECTION_NAME。

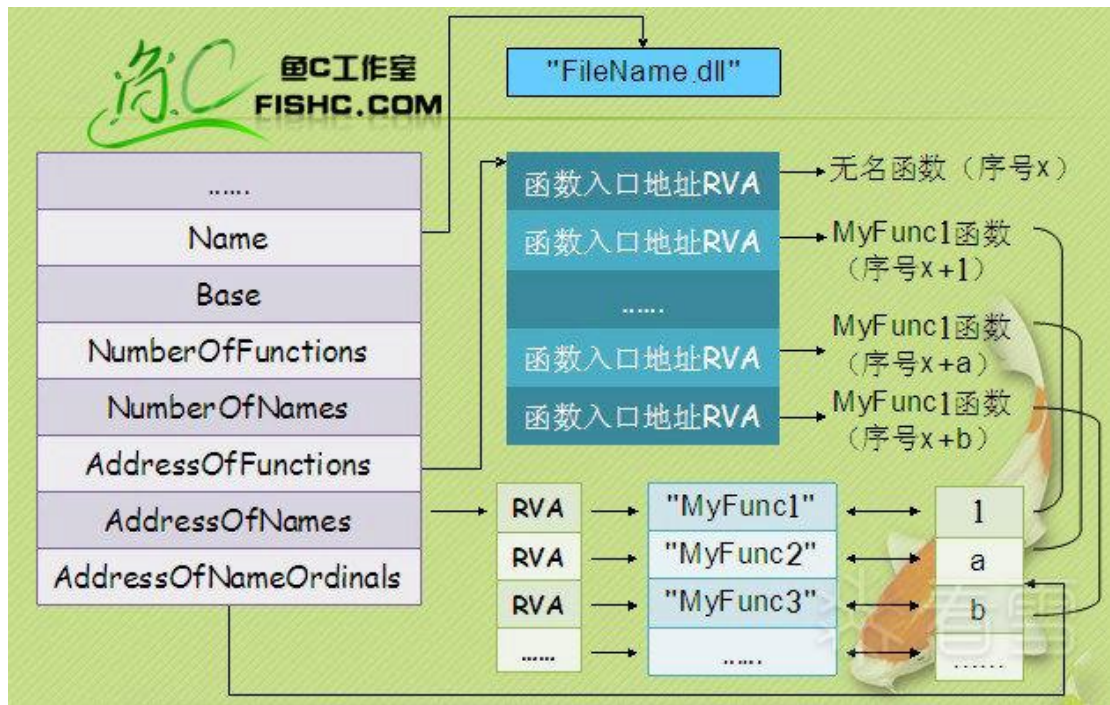
此时要注意的是 x64 和 x86 获取系统的信息稍微有所不同：

x86：ZwQuerySystemInformation 第一个参数是 SystemBasicInformation

2.6.2 GetProcAddress 实现

导出表有一个结构，是 IMAGE_EXPORT_DIRECTORY，主要注意里面的三个字段：

- 1、AddressOfNames
- 2、AddressOfNameOrdinals
- 3、AddressOfFunctions
- 4、NumberOfNames



代码实现的基本思路是：

循环 `NumberOfNames` 次，在 `AddressOfName` 地址里面查找 `targetFunc`，记住找到函数时的循环次数 `i`，通过 `AddressOfNameOrdinals` 找到函数的索引号，比如是 `b`，最后函数地址为 `AddressOfFunctions[b]`。

代码基本流程如下：

```
1 ImageExportDirectory=NtHeader.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress

1 AddressOfFunctions =MappingBaseAddress + ImageExportDirectory->AddressOfFunctions
2 );
3 AddressOfNames = MappingBaseAddress + ImageExportDirectory->AddressOfNames);
   AddressOfNameOrdinals =MappingBaseAddress+ImageExportDirectory->AddressOfNameOrdinals);
```

通过循环 `ImageExportDirectory->NumberOfNames` 次，比较名字是否和 `NtResumeThread` 相等，不等于循环，等于的话，就通过 `AddressOfNameOrdinals[i]` 得到函数索引 `FunctionOrdinal`，然后函数地址就等于 `FunctionAddress = (PVOID)((UINT8*)MappingBaseAddress + AddressOfFunctions[FunctionOrdinal])`

2.7 需要干掉 PageGuard

2、代码编写核心思路

Hook 函数 `nt!NtResumeThread` 的 SSDT 表，修改为 `KeBugCheckEx` 函数地址，然后 InLine HOOK 到 `FakeResumeThread` 函数去执行，最后执行 shellcode。

3、代码编写详细思路

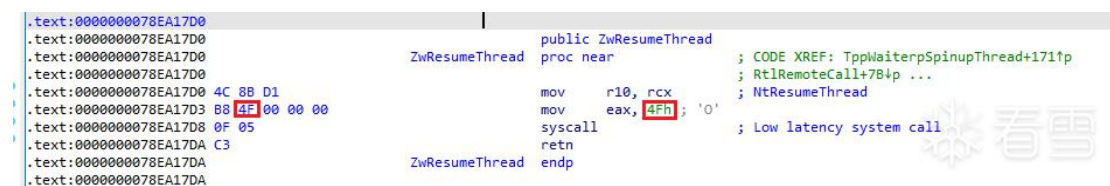
3.1 获取 SSDT 表地址

```
kd> rdmsr c0000082
msr[c0000082] = f ffff800`04038bc0
kd> uf ffff800`04038bc0
nt!KiSystemServiceCall64:
.....
nt!KiSystemServiceRepeat:
fffff800`03f07a72 4c8d1587be1f00 lea r10,[nt!KeServiceDescriptorTable
(fffff800`040be940)]
fffff800`03f07a79 4c8d1d40bf1f00 lea
r11,[nt!KeServiceDescriptorTableShadow (fffff800`040be980)]
```

通过 `PUCHAR StartSearchAddress = (PUCHAR)__readmsr(0xC0000082);` 作为起始地址，然后在 `PAGE_SIZE` 范围搜索特征码 (`4c\8d\15`)，然后地址加上偏移就是 SSDT 表地址了。

3.2 获取 `NtResumeThread` 函数在 SSDT 表里面的索引

在 `win7_sp1_x64` 系统，反汇编 `ntdll` 可知：



```
.text:0000000078EA17D0 public ZwResumeThread
.text:0000000078EA17D0 ZwResumeThread proc near
.text:0000000078EA17D0
.text:0000000078EA17D0 mov     r10, rcx
.text:0000000078EA17D3 B8 4F 00 00 00 mov     eax, 4Fh
.text:0000000078EA17D8 0F 05 syscall
.text:0000000078EA17DA C3 retn
.text:0000000078EA17DA ; CODE XREF: TppWaiterSpinupThread+171tp
.text:0000000078EA17DA ; RtlRemoteCall+7B+p ...
.text:0000000078EA17DA ; NtResumeThread
.text:0000000078EA17DA ; Low latency system call
.text:0000000078EA17DA ZwResumeThread endp
```

从函数地址偏移 4 个或 1 个字节得到此函数在 SSDT 表的索引。

3.3 计算原始函数地址

原始函数地址 = SSDT 基址 + 函数索引号得到的偏移地址>>4

```
1 __OldNtResumeThreadAddress =  
2 SSDTAddress->ServiceTableBase+SSDTAddress->ServiceTableBase[_FunctionIndexInSSDT]  
   >>4
```

3.4 HOOK SSDT 表

A、通过__readcr0 寄存器关闭写保护

B、InlineHook:把 KeBugCheck 的前面 14 个字节保存，然后改成跳转到 shellcode 执行。跳转汇编：

"\xFF\x25\x00\x00\x00\x00\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF";

C、计算 KeBugCheck 在 SSDT 表中基于 ServerTableBase 的偏移值，然后填入之前 NtResumeThread 的索引位置。

这样，执行 NtResumeThread 函数，实际先执行 KeBugCheck 函数，又由于 KeBugCheck 被 HOOK 了，在 KeBugCheck 函数里面执行 shellcode。

3.5 关于 FakeNtResumeThread

在这个函数里，我们主要做的事情如下：

- 1、通过 PsGetProcessImageFileName 函数，获取要注入进程的名字；
- 2、暴力搜索内存，获得 ntdll.dll 的基址，通过导出表获取 ldrLoadDLL 地址；
- 2、通过 KeStackAttachProcess 函数附加到进程；
- 3、通过 PsGetContextThread 函数获取进程的 rip
- 4、通过 PsSetContextThread 函数修改进程的 rip。

3.6 关于 shellcode

shellcode 结构体如下：

```
1 typedef struct _INJECT_DATA  
2 {  
3     CHAR ShellCode[0xA0];  
4     /*offset = 0xA0*/PWCHAR PathToFile;//LdrLoadDll 的第一个参数
```

```

5      /*offset = 0xA8*/ULONG64 DllCharacteristics;
6      /*offset = 0xB0*/PUNICODE_STRING pDllPath; //PUNICODE_STRING
7      DllPath
8      /*offset = 0xB8*/PHANDLE ModuleHandle; //Dll 句柄
9      /*offset = 0xC0*/ULONG64 AddrOfLdrLoadDll; //LdrLoadDll 地址
10     /*offset = 0xC8*/ULONG64 OriginalEIP; //原线程的 EIP
11     /*offset = 0xD0*/UNICODE_STRING usDllPath; //Dll 路径
12     /*offset = 0xE0*/WCHAR wDllPath[256]; //Dll 路径，也就是
        usDllPath 中的 Buffer
    }INJECT_DATA;

```

汇编代码如下：

```

push rax
push rax
push rax
push rbx
push rcx
push rdx
push rbp
push rsp
push rsi
push rdi
push r8
push r9
push r10
push r11
push r12
push r13
push r14
push r15
pushfq
call @Next
@Next:
pop rbx
and bx,0
mov rax,qword ptr ds:[rbx+0xc8]
xchg rax,qword ptr ds:[rsp+0x90]
sub rsp,0x30

```



```
mov r9,qword ptr ds:[rbx+0xb8]
mov r8,qword ptr ds:[rbx+0xb0]
mov rdx,qword ptr ds:[rbx+0xa8]
xor rcx,rcx
call [rbx+0xc0]
add rsp,0x30
popfq
pop r15
pop r14
pop r13
pop r12
pop r11
pop r10
pop r9
pop r8
pop rdi
pop rsi
pop rsp
pop rbp
pop rdx
pop rcx
pop rbx
pop rax
pop rax
ret
```

主要功能是：

- 1、加载要注入的 dll；
- 2、通过 `ret` 指令返回之前线程的 `rip`，跳转到之前线程的指令处执行；
- 3、重点是，`call` 函数的时候 `rsp` 要 16 字节对齐，否则会出现 `movaps` 错误；
- 4、之所以 `push3` 个 `rax`，一个是用于占位，一个是勇于让 `rsp` 实现 16 字节对齐，一个是保存 `rax` 寄存器。

4、关于代码

代码暂不上传。已经把关键点说得很详细了，我相信根据看雪 `x86` 的代码，你也能够写出 `x64` 系统下的代码了。当然，贴子如果精华了，或者需要的同学很多，再上传代码。