1、前言

今天的主题内容是 PsSetCreateProcessNotifyRoutineEx 的 DLL 注入,有一些关键点，很值得学习，还是老规矩，先说明几点：
1、今天是 win7_sp1_x64 平台下，同时注入 x86、x64、Wow64 程序；
2、重点不要放在代码上，要放在 windows 底层机制上，今天我会比之前的帖子说明底层机制更详细点，借用了老外的一些底层知识；
3、代码参考了《加密和解密》，在此感谢下先。因为不能直接在 VS2019 的 x64 位下编译，我修改了很多地方（shellcode 重新写了），至于改了其他哪些地方，我就不记得了，不过现在可以用 VS2019 直接编译运行。

2、注入的基本思路

通过 PsSetCreateProcessNotifyRoutineEx 的回调函数，获取目标进程的相关信息后，修改目标进程 ntdll.dll 中的 ZwTestAlert 的入口代码，在系统调用 ZwTestAlert 时，就会执行 shellcode，达到注入的目的。思路有了，现在要回答、解决 2 个关键问题：
1、为什么 HOOK 的是 ZwTestAlert 函数？系统加载 PE 流程是怎么样的？
2、ZwTestAlert 地址不可写，怎么改变其属性？当然可以用 ZwProtectVirtualMemory 函数，但问题是，你知道用哪个 ZwProtectVirtualMemory 函数吗？

3、前提知识

3.1 相比 SSDT HOOK 的优势

很简单，不需要干掉 page guard。这也是我要讲今天主题的一个原因。

3.2 PsSetCreateProcessNotifyRoutineEx 函数介绍

和这个 PsSetCreateProcessNotifyRoutineEx 函数相似的，还有一个函数，就是 PsSetCreateProcessNotifyRoutine，它们的区别是，前者可以阻止进程运行，后者只能监控进程的创建和退出。
PsSetCreateProcessNotifyRoutineEx 的回调函数声明为：

```
1 VOID MyCreateProcessNotifyRoutine(__inout PEPROCESS  Process, __in
  HANDLE  ProcessId, __in_opt PPS_CREATE_NOTIFY_INFO  CreateInfo)
```

基本用法是：

```
1 PsSetCreateProcessNotifyRoutineEx(MyCreateProcessNotifyRoutine,
  FALSE);
```

在回调函数里面获取目标进程的相关信息，HOOK ZwTestAlert 函数之后，就可以做你想做的事情了。

关于这个函数更详细的说明请参考下面网址：

https://www.cnblogs.com/priarieNew/p/9758980.html

2.3 x86、x64、wow64 代码兼容问题

牢牢记住，x64 系统，驱动使用的是 64 位编译器；x86 系统，驱动使用的是 x86 编译器；但是 wow64 程序，驱动使用的是 64 位编译器，环境在 64 位下，但注入的程序却是 32 位的，所以得出以下结论：

A、x86、x64 的注入代码可以写到一起，只是声明变量的时候，要使用 PVOID、SIZE_T、ULONG_PTR 等来定义，这样不管是使用 x64 编译器，还是 x86 编译器，变量大小都可以兼容；

B、wow64 的注入代码，声明变量的时候使用 x86 的常规声明，不能混入原生 x86、x64 系统，因为 wow64，必须使用 x64 编译器，但变量声明却是 32 位的。

4、关键问题解决

4.1 为什么 HOOK 的是 ZwTestAlert 函数

4.1.1 PE 加载流程

我们知道，PE 的加载流程是：

a、调用 CreateProcess 启动进程

b、生产内核对象，分配 4GB 进程空间

c、加载 ntdll.dll

d、创建进程的主线程;

e、主线程初始化输入表，系统加载器加载 dll 文件以及完成重定位、IAT

f、主线程为每个 DLL 调_DLLMainCRTStartup 函数

g、主线程根据 EXE 是 GUI（图形化）程序还是 CUI（控制台）程序执行

MainCRTStartup，
h、最后一步执行 WinMain(或者 main 函数)
知道了 PE 加载流程，我们就知道我们的 HOOK 大概在哪个位置了，下面的流程是从创建进程的主线程那里开始分析。

4.1.2 ZwTestAlert 函数执行时机

先来点看起来高大上的东西：
when the first thread in process start - this is special case. need do many extra jobs for process initialization. at this time only two modules loaded in process - EXE and ntdll.dll . LdrInitializeThunk call LdrpInitializeProcess for this job. if very briefly:

1. different process structures is initialized
2. loading all DLL (and their dependents) to which EXE statically linked - but not call they EPs !
3. called LdrpDoDebuggerBreak - this function look - are debugger attached to process, and if yes - int 3 called - so debugger receive exception message - STATUS_BREAKPOINT - most debuggers can begin UI debugging only begin from this point. however exist debugger(s) which let as debug process from LdrInitializeThunk - all my screenshots from this kind debugger
4. important point - until in process executed code only from ntdll.dll (and may be from kernel32.dll) - code from another DLLs, any third-party code not executed in process yet.
5. optional loaded shim dll to process - Shim Engine initialized. but this is OPTIONAL
6. walk by loaded DLL list and call its EPs with DLL_PROCESS_DETACH
7. TLS Initializations and TLS callbacks called (if exists)
8. ZwTestAlert is called - this call check are exist APC in thread queue, and execute its. this point exist in all version from NT4 to win 10. this let as for example create process in suspended state and then insert APC call (QueueUserAPC) to it thread (PROCESS_INFORMATION.hThread) - as result this call will be executed after process will be fully initialized, all DLL_PROCESS_DETACH called, but before EXE entry point. in context of first process thread.
9. and NtContinue called finally - this restore saved thread context and we finally jump to thread EP.

上面是对 PE 加载流程一些更详细的说明，我来翻译下：

当进程中的第一个线程创建时，进程还需要做很多初始化工作。此时进程中只加载了两个模块，EXE 和 ntdll.dll。LdrInitialize Thunk 为此调用 LdrpInitalizeProcess（这就是第一篇贴子说的，Ldr 初始化未完成的情况），总体流程如下：
1、初始化进程各种结构；
2、加载 EXE 静态链接的所有 dll 及 dll 的依赖项，但这不是 EP（Entry Point）；
3、调用 LdrpDoDebuggerBreak：判断是否有调试器附加进程。如果有，则执行 int 3 指令，让调试器接收 STATUS_BREAKPOINT 的异常消息。多数带 UI 界面的调试器就是从这里开始调试的。但是存在一些调试器，也可以让调试从 LdrInitializeThunk 开始。
4、 很重要的一点：ntdll.dll（也可能 kernel32.dll）中的代码执行完之前，其他任何 dll 的和第三方库的代码不会被执行；
5、Windows 兼容性引擎初始化，加载兼容性 dll（如果有）。（ShinEngHOOK 在此处）
6、加载 DLL，并执行 DLL 中的 DLL_PROCESS_DETACH；
7、TLS 初始化并且执行 TLS 的回调函数（如果有）（一些反调试就在此处）；
8、调用 ZwTestAlert 函数。这个函数会检查线程的 APC 队列是否有函数，如果有，则执行这些函数。这个机制从 NT4 到 win10 版本的操作系统都存在。如果以 suspended 状态创建进程，然后把 APC 函数插入到线程，在进程初始化完成后，将会执行这些 APC 函数。在到达 EXE 的 EP 点之前，所有的 DLL_PROCESS_DETACH 都会被调用。（这儿可以 APC HOOK）
9、最后调用 NtContinue 函数，这将保存线程的 context ，并且到达线程的 EP 点。

我觉得，这儿的内容是今天晚上介绍这篇 HOOK 的最重要的。因为，从这个流程入手，你至少明白了以下几点：
1、ShinEngHOOK 的时机；
2、TLS 反调试的时机；
3、APC HOOK 的时机；
4、ZwTestAlert 函数被 HOOK 的原因。
5、发散思维，当然也可以 HOOK NtConitnue 函数。

4.2 哪个 ZwProtectVirtualMemory 函数

我在这里吃了大亏，希望你别像我一样。
在 ntdll 里面，Zw *和 Nt* 函数都是一样的，都在同一个地址，都同时存在于导出表（比如 ZwReadFile、NtReadFile 都在导出表，都在同一个地址， 而且他们最终都会调用到 ntoskrnl 中的 NtReadFile 中去），但是 ntoskrnl.exe 导出的 ZwReadFile 和 NtReadFile 却是不同的。

Ntoskrnl 导出的 NtReadFile 是真正的执行函数 而 ZwReadFile 仍然是一个 stub 函数。
内核态调用 ZwProtectVirtualMemory 会将 Previous Mode 设置为 Kernel Mode 然后再调用到 NtProtectVirtualMemory 中。而在内核态直接调用 NtProtectVirtualMemory 不会改变 Previous Mode。而在 NtProtectVirtualMemory 中会检测当前调用来自用户态还是内核态如果是来自内核态不会检测参数，而如果是来自用户态，就会做一系列的参数检测。而我们知道内核组件可能运行在任意进程的上下文中。当它调用 NtProtectVirtualMemory 时，因为 Previous Mode 很可能是 User Mode，而我们的参数请求的内核态的地址，这时通常就会产 STATUS_ACCESS_VIOLATION
错误。
说了这么多，意思就是：
1、在内核态，不能使用 ntdll.dll 里面的 ZwProtectVirtualMemory\NtProtectVirtualMemory
2、在内核态，也不能使用 ntoskrnl.exe 导出的 NtProtectVirtualMemory
3、在内核态，要使用 ntoskrnl.exe 导出的 ZwProtectVirtualMemory

4.3 Zw 函数流程总结

1、Zw 函数会在 KiSystemService 中将 ETHREAD 中的 PreviousMode 改为 KernelMode，最后在 Nt 函数中如果是 KernelMode 就会跳过对参数是否可写的验证，如果是 UserMode 就会验证。如果是 UserMode，访问内核地址会报错，所以如果内核中直接调用 Nt 函数，需要手动将 PreviousMode 修改为 KernelMode，否则无法访问内核地址，而修改 PreviousMode 并且通过系统服务表获取 SSDT 函数这个过程是很复杂的，直接调用内核导出的 Zw 函数就行，不过在调用 Zw 函数的时候需要自己对地址的可写性验证。而且通过 Zw 函数调用会在系统空间堆栈上有个属于本次调用的自陷框架。
2、32 位下 Zw 函数会将内核模式保存在 CS 最后一位上，调用 KiSystemService 修改 PreviousMode 为 KernelMode，接着跳转到 KiFastCallEntry 中间的地方，初始化一些寄存器，最后通过 call ebx 的方式调用 Nt 函数，最后通过 KiSystemExit 返回。
3、64 位下 Zw 函数会调用 KiServiceInternal，在这个函数中修改 PerviousMode 为 KernelMode，然后跳转到 KiSystemCall64 中的 KiSystemServiceStart 部分，接着在 KiSystemServiceRepeat 部分通过 jmp r11 调用 Nt 函数，最后通过 KiSystemServiceExit 函数返回。

4.4 如何获取 ntoskrnl.exe 中的未导出函数

ZwProtectVirtualMemory

A、映射 ntdll.dll 到 ring0 空间

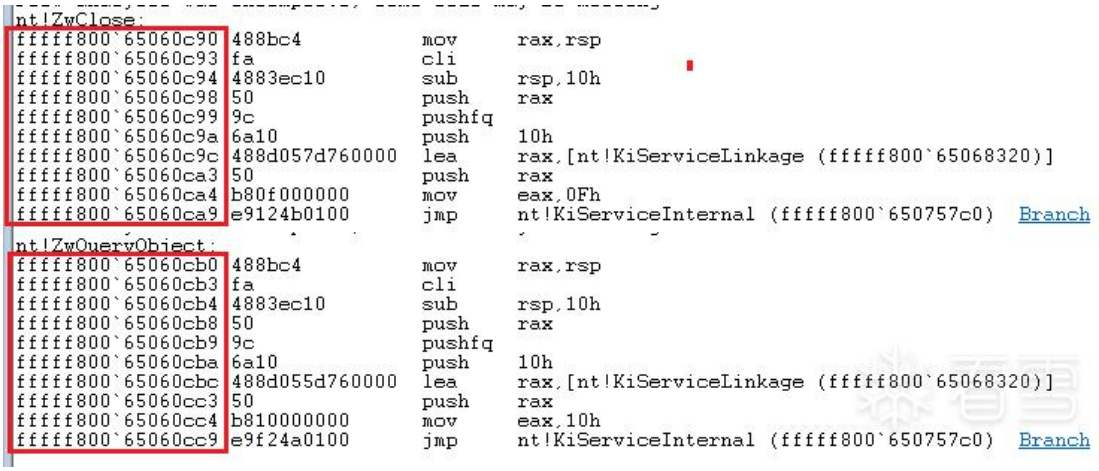
B、获取 ZwprotectVirtualMemory 在 SSDT 表中的 Index；

获取方法，见：

【DLL 注入编写与分析系列之一】x64 平台 SSDT HOOK 之 NtResumeThread 注入

C、获取 ZwprotectVirtualMemory 函数地址

看来看看 ntoskrnl.exe 中函数的布局：



由图可知，两个 Zw 函数之间的长度大小是一样的，所以，可以这样计算：

A、获取 ZwClose 和 ZwOpenProcess 函数的 Index（ZwProtectVirtualMemory 无法这样获取,因为没有导出）

```
1 #ifdef _WIN64
2 #define GetSysCallIndexFromKernel(fun) (*(LONG*)((BYTE*)fun +
3 0x15))
4 #else
5 #define GetSysCallIndexFromKernel(fun) (*(LONG*)((BYTE*)fun + 1))
  #endif

1 IndexOfClose = GetSysCallIndexFromKernel(ZwClose);
2 IndexOfOpenProcess = GetSysCallIndexFromKernel(ZwOpenProcess);
```

B、通过 ZwClose-ZwOpenProces 除以 Index 之间的差值，就得到每个函数的大小 FuncSize

C、通过 ZwClose-FuncSize*IndexofClose 就得到 Zw 系列函数的 BaseAddress*

*D、然后：BaseAddress+Funcsize*IndexOfZwProtectVirtualMemory 就得到

ZwProtectVirtualMemory 函数地址了。
之所以能够这样利用，就是因为：每个 Zw 函数之间的大小是相等的。

5、代码编写详细思路

A、首先获取 ZwProtectVirtualMemory 函数的地址；
B、设置回调函数 MyCreateProcessNotifyRoutine：

```
1 PsSetCreateProcessNotifyRoutineEx(MyCreateProcessNotifyRoutine,
  FALSE);
```

C、判断是否是 Wow64 位程序
通过

```
1 status = ZwQueryInformationProcess(hProcess,
  ProcessWow64Information, &Wow64Info, sizeof(ULONG_PTR),
  &uReturnLen);
```

wow64 加载，SysWow64\ntdll.dll，其他则加载 System32\ntdll.dll
D、通过 ZwProtectVirtualMemory 函数修改 ZwTestAlert 函数地址的属性
E、编写 shellcode
x86/x64shellcode：
对驱动来说，编译 32 位的驱动就只能在 32 位系统运行，编译 64 位的驱动就只能在 64 位系统运行，所以对原生系统来说，编译的时候是分别用 32 位、64 位编译，因此它们的 shellcode 可以写到一个函数里面，用类似 SIZE_T、ULONG_PTR 的定义兼容 32 位、64 位。
wow64 sehllcode：
要注入 wow64 位程序，说明该程序依然是运行在 64 位程序上，驱动依然是 64 位的，编译的时候就要用 64 位编译。但是目标进程却是 32 位的，因此，shellcode 里面的定义就要用 32 位的定义，比如 ULONG 之类的定义，去注入 32 位程序。没办法三种情况一起兼容。对驱动来说，编译 32 位的驱动就只能在 32 位系统运行，编译 64 位的驱动就只能在 64 位系统运行，不存在 32 位运行在 64 位的情况。
Shellcode 汇编关键思路：
1、恢复 ZwTestAlert 入口地址；
2、调用 ZwProtectVirtualMemory 恢复 ZwTestAlert 地址属性
3、调用 LdrLoadDll 注入 Dll

shellcode 是我重新写的。

push rax //（确保 rsp 16 字节对齐，下面 call 时，Movaps 指令 16 字节对齐）

```
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//(定位到 shellcode 开头)
mov rcx,rbx
call @LoadDllAndRestoreExeEntry
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
sub dword ptr ds:[rsp],6
ret
@LoadDllAndRestoreExeEntry:
mov rdi,qword ptr ds:[rbx+0xF8]
lea rsi,qword ptr ds:[rbx+0x108]
xor rcx,rcx
add rcx,2
rep movs qword ptr ds:[rdi],qword ptr ds:[rsi]
lea rax,qword ptr ds:[rbx+0xE8]
mov qword ptr ds:[rsp+0x20],rax
mov r9,qword ptr ds:[rbx+0xE8]
lea r8,qword ptr ds:[rbx+0xF0]
lea rdx,qword ptr ds:[rbx+0xE0]
or rcx,0xFFFFFFFFFFFFFFFF
call qword ptr ds:[rbx+0x100]//用 ZwProtectVirtualMemory
lea r9,[rbx+0xC0]
mov r8,[rbx+0xC8]
mov rdx,[rbx+0xD0]
xor rcx,rcx
jmp qword ptr ds:[rbx+0xD8]
```


F、构造 call 指令，跳转到 shellcode
x64:FF 15
x86:E8

6 代码

贴子变优了，或者加精了，或者要的同学多，再上传代码吧。