Windows Keylogger Part 2: Defense against user-land

By Published by Eye of Ra View all posts by Eye of Ra

Published: 2017-06-27 · Archived: 2026-04-06 01:48:38 UTC

Now, this is the interesting part. Recall from part 1, I had showed you 4 hooking methods using in Windows user-mode and today we will analyze each of them for answering one question: how to detect it? Let's see!

Windows test machine:

> Windows 7 x86: version 6.1.7601.17514 Service Pack 1 Build 7601)
>
> ntoskrnl.exe: 6.1.7601.17514 (win7sp1_rtm.101119-1850), md5: 2088D9994332583EDB3C561DE31EA5AD
>
> win32k.sys: 6.1.7601.17514 (win7sp1_rtm.101119-1850), md5: 687464342342B933D6B7FAA4A907AF4C

*All offset values and structures I used in this part are from test machine.

Windows Hooking: SetWindowsHookEx

When we register hook using SetWindowsHookEx, the system saves our hook procedure in a hook chain which is a list of pointers. Because we can register many type of message WH_* so there will be a hook chain for each type of message. Therefore our targets are:

- The location of hook chains in system's memory (for WH_KEYBOARD, WH_KEYBOARD_LL message type)
- How to find the process name of the found hook

For the location of hook chain, I have a magic string:

```
1 nt!_ETHREAD + 0x0 => nt!_KTHREAD + 0x088 => nt!_TEB + 0x40 => win32k!tagTHREADINFO + 0xCC => win32k!tagDESKTOPINFO + 0x10 => win32k!tagHOOK
```

Every structure is clear (thanks for windows symbols :d). Offset values are of my test machine and can be vary on each Windows version and build number (ntoskrnl and win32k.sys)

From nt!_ETHREAD, it must be **a GUID thread**. We can get GUI thread from "explorer.exe" or create a thread for your own.

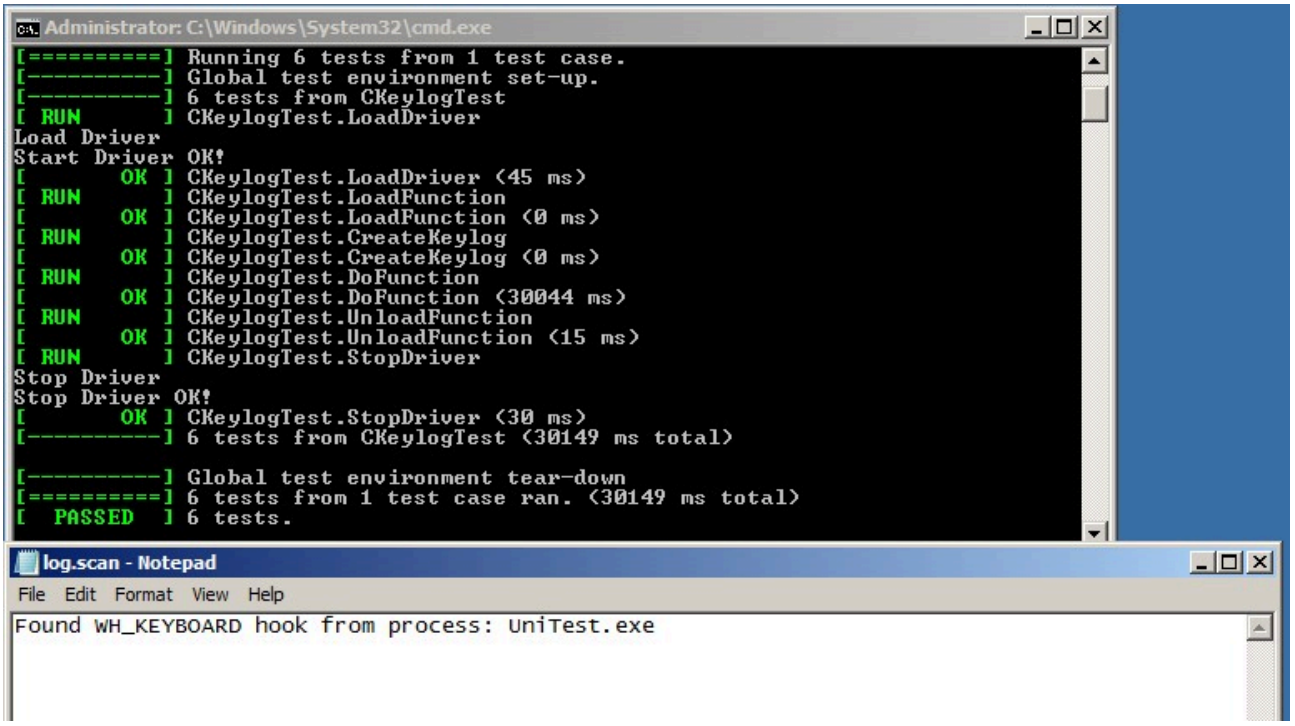
At the end of magic string, we get a final location of all global hook chains in the system. This is a array pointers of tagHOOK with **16 entries**, the index of array is the value of WH_* message type (actually **index = WH_* + 1**). If entry is not NULL then we found a global hook chain.

```
kd> dt win32k!tagHOOK
+0x000 head             : _THRDESKHEAD
+0x014 phkNext          : Ptr32 tagHOOK
+0x018 iHook            : Int4B
+0x01c offPfn           : Uint4B
+0x020 flags            : Uint4B
+0x024 ihmod            : Int4B
+0x028 ptiHooked        : Ptr32 tagTHREADINFO
+0x02c rpdesk           : Ptr32 tagDESKTOP
+0x030 nTimeout         : Pos 0, 7 Bits
+0x030 fLastHookHung    : Pos 7, 1 Bit
dt win32k!_THRDESKHEAD
+0x000 h                : Ptr32 Void
+0x004 cLockObj         : Uint4B
+0x008 pti              : Ptr32 tagTHREADINFO
+0x00c rpdesk           : Ptr32 tagDESKTOP
+0x010 pSelf            : Ptr32 UChar
```

From _THRDESKHEAD in tagHook we get tagTHREADINFO of the process that set the hook. So we can get process id then process name. Here is another magic string

```
processIdOfHooker = PsGetProcessId(IoThreadToProcess((PETHREAD)(*pCurHook->head.pti)));
```

The scan result:



Okay, that's all we need for hunting the global hook of Windows message. Oh, what about local hook? :d

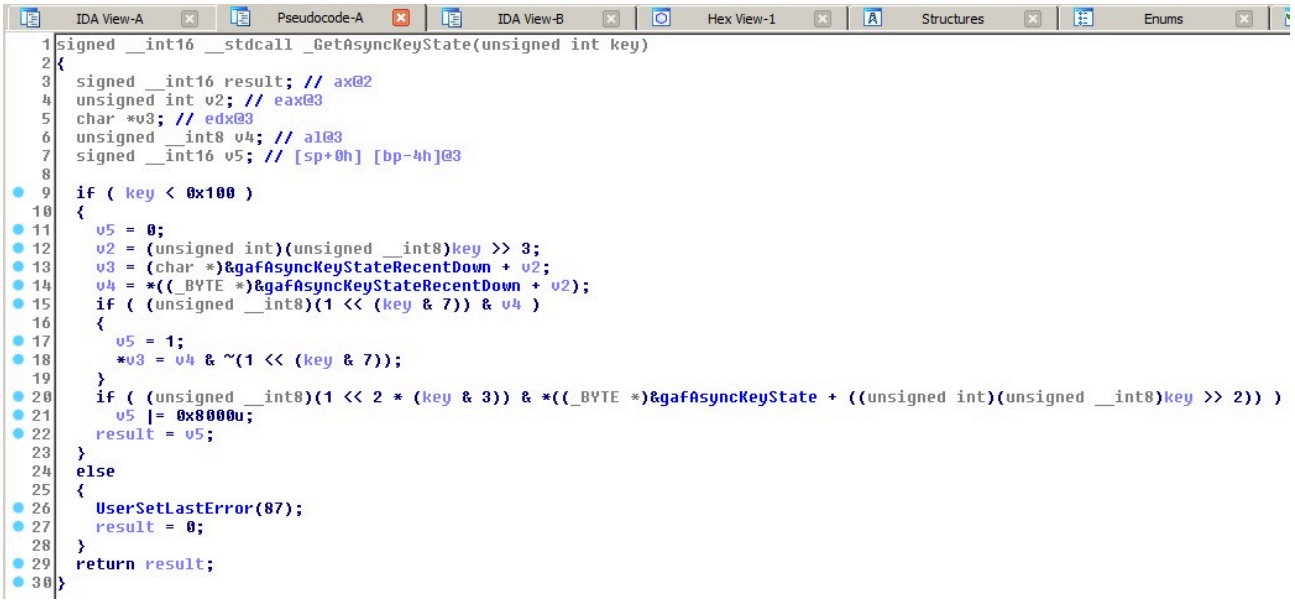
Here is the magic string for local hook:

```
1  nt!_ETHREAD + 0x0 => nt!_KTHREAD + 0x088 => nt!_TEB + 0x40 => win32k!tagTHREADINFO + 0x198
   =>  win32k!tagHOOK
```

Quite similar to global hook but you can see the location of local hook chains is in tagTHREADINFO structure of process and it will be local at that process. The hook chains in tagDESKTOPINFO is global for all process in the same desktop.

Windows Polling

Well, I actually don't have any idea for scanning this type of hooking. Why? Because it reads directly keys state from the internal structure and seems there is no way to check who is reading that.



How about API hooking for GetAsyncKeyState(), GetKeyboardState()? Yes, we can detect with API hooking but I don't like it because a global API hooking for all processes in the system is not a good idea. Using API hooking, we can check the frequency and range of checked keys for keylogging detection.

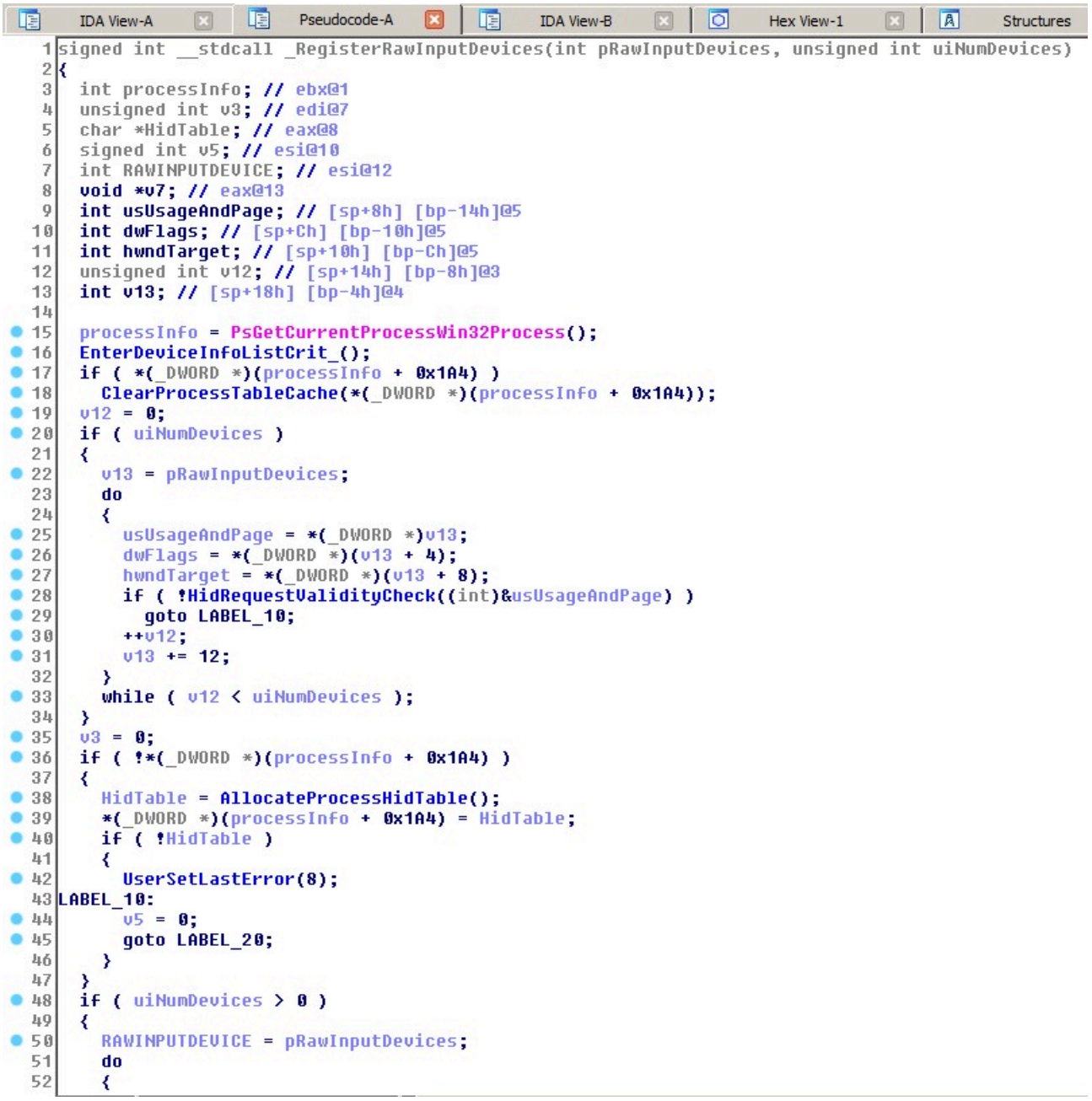
Raw Input

We start analyzing with **RegisterRawInputDevices()** function in user32.dll. From this API, it will call **NtUserRegisterRawInputDevices()** in win32k.sys

IDA View-A | Pseudocode-A | IDA View-B | Hex View-1 | Structures

```
1  int __stdcall NtUserRegisterRawInputDevices(size_t pRawInputDevices, int uiNumDevices, int cbSize)
2  {
3    size_t v3; // ebx@1
4    int v4; // esi@2
5    int v5; // eax@6
6    PVOID v6; // edi@14
7    char v8; // [sp+10h] [bp-2Ch]@15
8    PVOID v9; // [sp+1Ch] [bp-20h]@1
9    int v10; // [sp+20h] [bp-1Ch]@13
10   CPPEH_RECORD ms_exc; // [sp+24h] [bp-18h]@4
11
12   v9 = 0;
13   UserEnterUserCritSec();
14   v3 = pRawInputDevices;
15   if ( pRawInputDevices && (v4 = uiNumDevices) != 0 && cbSize == 0xC )
16   {
17     ms_exc.registration.TryLevel = 0;
18     if ( (unsigned int)uiNumDevices > 0x15555555 )
19       ExRaiseAccessViolation();
20     v5 = 12 * v4;
21     if ( 12 * v4 )
22     {
23       if ( v3 & 3 )
24         ExRaiseDatatypeMisalignment();
25       if ( v5 + v3 > W32UserProbeAddress || v5 + v3 < v3 )
26         *(_BYTE *)W32UserProbeAddress = 0;
27     }
28     if ( UIntMult(v4, 0xCu, (int)&pRawInputDevices) >= 0 )
29     {
30       v6 = Win32AllocPoolWithQuota(pRawInputDevices, 0x79737355u);
31       v9 = v6;
32       if ( !v6 )
33         ExRaiseStatus(0xC0000017);
34       PushW32ThreadLock(v6, &v8, ExFreePool);
35       memcpy(v6, (const void *)v3, pRawInputDevices);
36       ms_exc.registration.TryLevel = -2;
37       v10 = _RegisterRawInputDevices((int)v6, v4);
38       if ( v9 )
39         PopAndFreeAlwaysW32ThreadLock(&v8);
40     }
41     else
42     {
43       v10 = 0;
44       UserSetLastError(87);
45       ms_exc.registration.TryLevel = -2;
46     }
47   }
48   else
49   {
50     v10 = 0;
51     UserSetLastError(87);
52   }
```

After some checks and synchronizes, we go to **_RegisterRawInputDevices()**




```
signed int __stdcall _RegisterRawInputDevices(int pRawInputDevices, unsigned int uiNumDevices)
{
  int processInfo; // ebx@1
  unsigned int v3; // edi@7
  char *HidTable; // eax@8
  signed int v5; // esi@10
  int RAWINPUTDEVICE; // esi@12
  void *v7; // eax@13
  int usUsageAndPage; // [sp+8h] [bp-14h]@5
  int dwFlags; // [sp+Ch] [bp-10h]@5
  int hwndTarget; // [sp+10h] [bp-Ch]@5
  unsigned int v12; // [sp+14h] [bp-8h]@3
  int v13; // [sp+18h] [bp-4h]@4

  processInfo = PsGetCurrentProcessWin32Process();
  EnterDeviceInfoListCrit_();
  if ( *(_DWORD *)(processInfo + 0x1A4) )
    ClearProcessTableCache(*(_DWORD *)(processInfo + 0x1A4));
  v12 = 0;
  if ( uiNumDevices )
  {
    v13 = pRawInputDevices;
    do
    {
      usUsageAndPage = *(_DWORD *)v13;
      dwFlags = *(_DWORD *)(v13 + 4);
      hwndTarget = *(_DWORD *)(v13 + 8);
      if ( !HidRequestValidityCheck((int)&usUsageAndPage) )
        goto LABEL_10;
      ++v12;
      v13 += 12;
    }
    while ( v12 < uiNumDevices );
  }
  v3 = 0;
  if ( !*(_DWORD *)(processInfo + 0x1A4) )
  {
    HidTable = AllocateProcessHidTable();
    *(_DWORD *)(processInfo + 0x1A4) = HidTable;
    if ( !HidTable )
    {
      UserSetLastError(8);
LABEL_10:
      v5 = 0;
      goto LABEL_20;
    }
  }
  if ( uiNumDevices > 0 )
  {
    RAWINPUTDEVICE = pRawInputDevices;
    do
    {
```

```
53      v7 = (void *)SearchProcessHidRequest(
54                     processInfo,
55                     *(_WORD *)RAWINPUTDEVICE,
56                     *(_WORD *)(RAWINPUTDEVICE + 2),
57                     &pRawInputDevices);
58      if ( *(_BYTE *)(RAWINPUTDEVICE + 4) & 1 )
59      {
60        if ( v7 )
61          FreeHidProcessRequest(v7, pRawInputDevices, *(_DWORD *)(processInfo + 0x1A4));
62      }
63      else if ( !SetProcDeviceRequest(processInfo, RAWINPUTDEVICE, v7, pRawInputDevices) )
64      {
65        goto LABEL_10;
66      }
67      ++v3;
68      RAWINPUTDEVICE += 12;
69    }
70    while ( v3 < uiNumDevices );
71  }
72  v5 = 1;
73LABEL_20:
74  if ( *(_DWORD *)(processInfo + 0x1A4) )
75  {
76    AdjustLegacyDeviceFlags(processInfo);
77    FixupOrphanedExclusiveRequests(processInfo);
78    CleanupFreedTLCInfo();
79    HidDeviceStartStop();
80  }
81  LeaveDeviceInfoListCrit_();
82  return v5;
83}
```

That’s quite clear at here. **PsGetCurrentProcessWin32Process()** return win32k!tagPROCESSINFO structure. It check something at offset **0x1A4**, using windbg we have:

```
1  kd> dt win32k!tagPROCESSINFO
2  +0x000 Process          : Ptr32 _EPROCESS
3  …
4  +0x1a4 pHidTable        : Ptr32 tagPROCESS_HID_TABLE
5  +0x1a8 dwRegisteredClasses : Uint4B
6  +0x1ac pvwplWndGCList   : Ptr32 VWPL
```

A pointer to win32k!tagPROCESS_HID_TABLE. Interesting!!!

The lines in range 20-34 validate the registration data (that will be called HID Request)

The lines in range 36-47 allocate HID Table if it not exist. That means if tagPROCESSINFO->pHidTable is null, no raw input was be registered in this process.

The lines in range 48-71 set HID request into HID table

The remaining of function is update the flags and restart HID device

Let’s see the function **SetProcDeviceRequest()**

IDA View-A | Pseudocode-A | IDA View-B | Hex View-1 | Structures

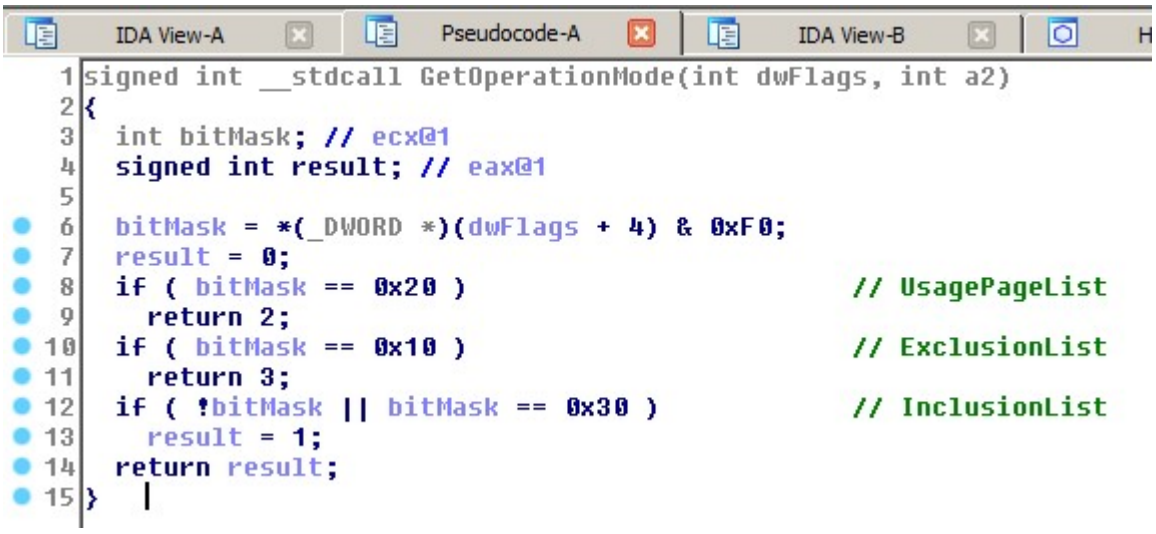
```
int __stdcall SetProcDeviceRequest(int processInfo, int RAWINPUTDEVICE, void *a3, int a4)
{
  int v4; // esi@1
  int v5; // eax@1
  unsigned int v6; // ecx@1
  int isLegacyDev; // ebx@1
  int result; // eax@2
  void *hidRequest; // edi@5
  int operationMode; // [sp+8h] [bp-4h]@5
  int RAWINPUTDEVICEa; // [sp+18h] [bp+Ch]@2

  v4 = RAWINPUTDEVICE;
  v5 = IsLegacyDevice(*(_WORD *)RAWINPUTDEVICE, *(_WORD *)(RAWINPUTDEVICE + 2));
  v6 = *(_DWORD *)(RAWINPUTDEVICE + 8);
  isLegacyDev = v5;
  if ( v6 )
  {
    result = ValidateHwnd(v6);
    RAWINPUTDEVICEa = result;
    if ( !result )
      return result;
  }
  else
  {
    RAWINPUTDEVICEa = 0;
  }
  hidRequest = a3;
  operationMode = GetOperationMode(v4, isLegacyDev);
  if ( a3 )
  {
    RemoveProcRequest(processInfo, a3, a4, isLegacyDev);
  }
  else
  {
    hidRequest = AllocateHidProcessRequest(*(_WORD *)v4, *(_WORD *)(v4 + 2));
    if ( !hidRequest )
    {
      UserSetLastError(8);
      return 0;
    }
  }
  if ( !InsertProcRequest(processInfo, v4, (int)hidRequest, operationMode, isLegacyDev, RAWINPUTDEVICEa) )
  {
    if ( hidRequest )
      ExFreePoolWithTag(hidRequest, 0);
    return 0;
  }
  if ( isLegacyDev )
    SetLegacyDeviceFlags(*(_DWORD *)(processInfo + 0x1A4), v4);
  return 1;
}
```

The system allocates a HID Request and insert it to HID Table

```
kd> dt win32k!tagPROCESS_HID_TABLE
+0x000 link : _LIST_ENTRY
+0x008 InclusionList : _LIST_ENTRY
+0x010 UsagePageList : _LIST_ENTRY
+0x018 ExclusionList : _LIST_ENTRY
+0x020 spwndTargetMouse : Ptr32 tagWND
+0x024 spwndTargetKbd : Ptr32 tagWND
```

There are 3 lists of HID Request that were used for raw input: InclusionList, UsagePageList and ExclusionList. To which list will be inserted, it depends on dwFlags value of tagRAWINPUTDEVICE structure when we call RegisterRawInputDevices();



With keylogger, we using **RIDEV_NOLEGACY | RIDEV_INPUTSINK** flags therefore the list will be **InclusionList**.

The last structure we concerned is win32k!tagPROCESS_HID_REQUEST

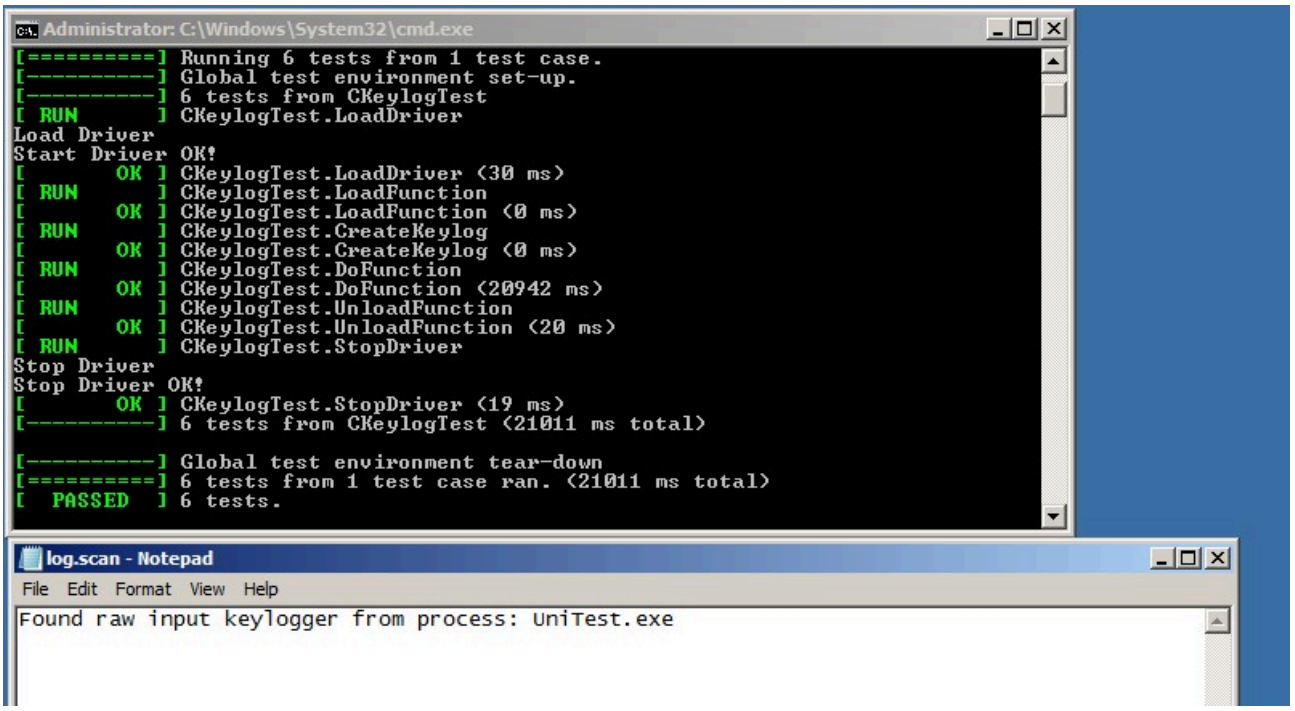
```
1   kd> dt win32k!tagPROCESS_HID_REQUEST
2   +0x000 link : _LIST_ENTRY
3   +0x008 usUsagePage : Uint2B
4   +0x00a usUsage : Uint2B
5   +0x00c fSinkable : Pos 0, 1 Bit
6   +0x00c fExSinkable : Pos 1, 1 Bit
7   +0x00c fDevNotify : Pos 2, 1 Bit
8   +0x00c fExclusiveOrphaned : Pos 3, 1 Bit
9   +0x010 pTLCInfo : Ptr32 tagHID_TLC_INFO
10  +0x010 pPORequest : Ptr32 tagHID_PAGEONLY_REQUEST
11  +0x010 ptr : Ptr32 Void
12  +0x014 spwndTarget : Ptr32 tagWND
```

We can see usUsagePage, usUsage and spwndTarget are the params in tagRAWINPUTDEVICE.

Bingo!!! For raw input detection we will:

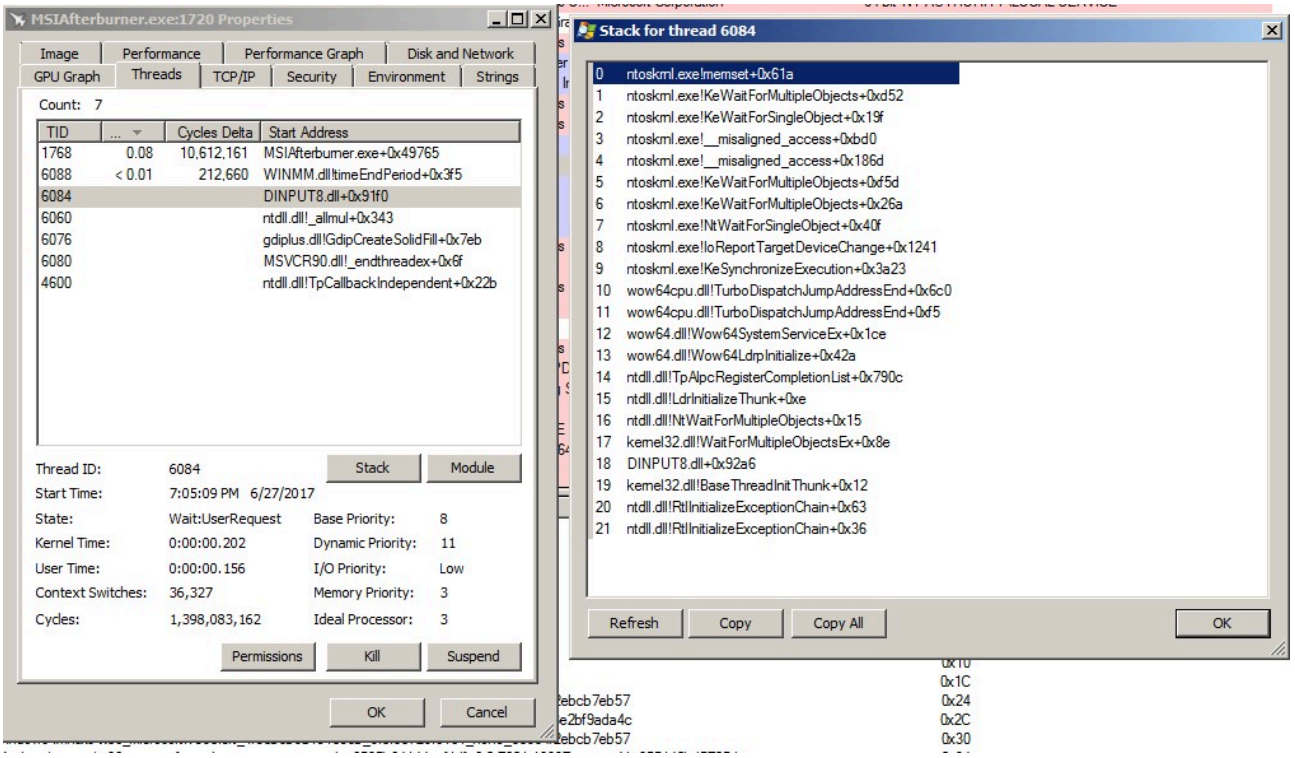
1. Enumerate all process in the system
2. With each process, we will traverse pID -> PEPROCESS -> tagPROCESSINFO -> tagPROCESS_HID_TABLE -> tagPROCESS_HID_REQUEST
3. If we found an entry with usUsagePage = 1 (generic desktop controls) and usUsage = 6 (keyboard) then this process is using raw input keylog.

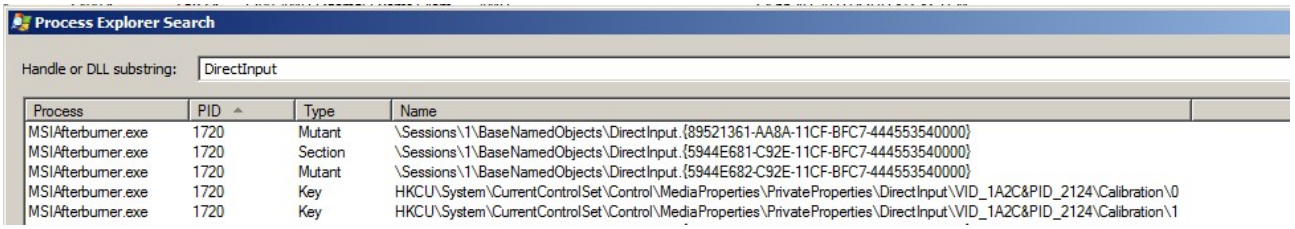
The scan result:



Direct Input

When checking direct input, I found some interesting signatures in the process registering the hook.



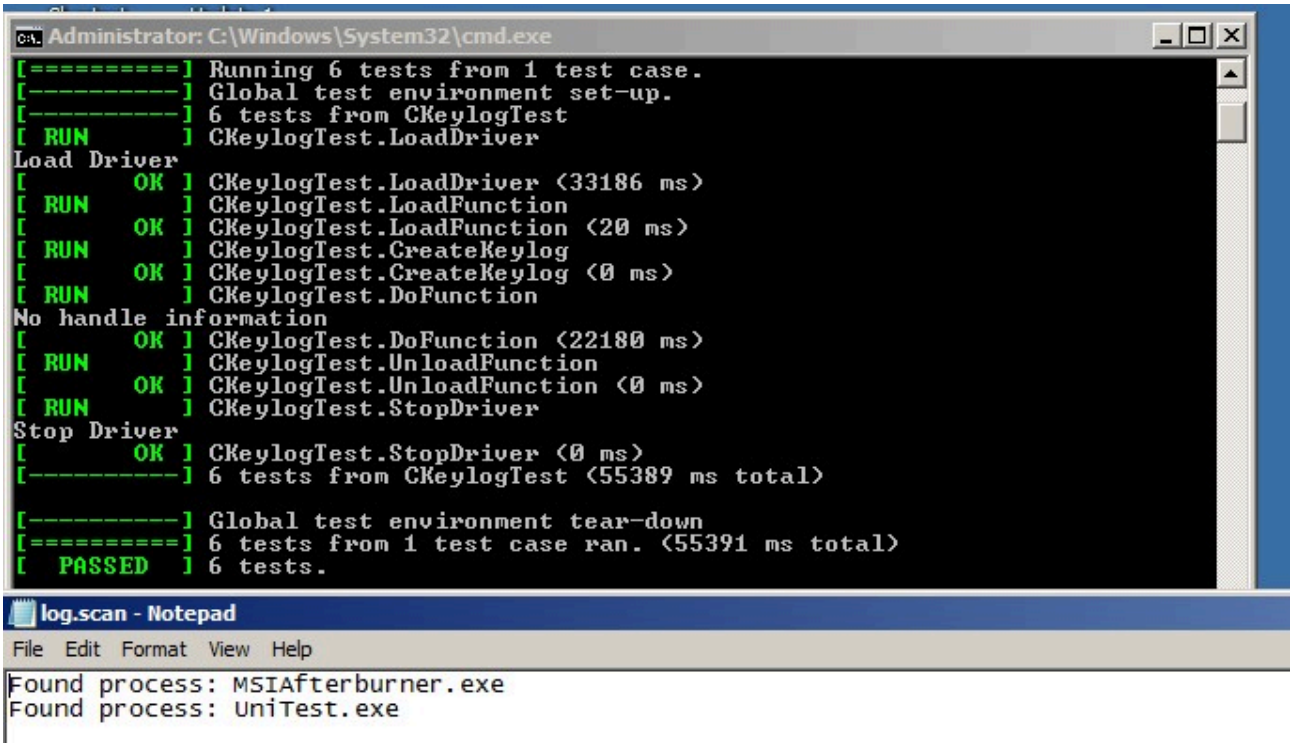


With MSIAfterburner.exe, we found some handles related to direct input (**Mutant, Section, Key**). From running threads, we also found a thread of **DINPUT8.dll** (a library of Microsoft DirectInput).

Done! For direct input detection we will:

1. Enumerate all processes in the system
2. With each process, enumerate all mutant, section, key that match the handle signatures.
3. If all handle signatures matched, we get start address of all threads in that process. If start address is in the address range of DINPUT8.DLL then we found the direct input keylog.

The scan result:



Conclusion

We have a summary table for scanning user-mode keylogger:

	Scan method	**Scan from**
Windows Hooking (SetWindowsHookEx)	Structure scanning	Kernel-mode
Windows Polling	API hooking	
Raw Input	Structure scanning	Kernel-mode
Direct Input	Signature scanning	User-mode (Admin required)

Source: https://eyeofrablog.wordpress.com/2017/06/27/windows-keylogger-part-2-defense-against-user-land/