# A bit away from Kernel execution

A ‘write-what-where’ kernel memory overwrite tale.

By ar1vr (01/12/2011)
picturoku.blogspot.com

Good arbitrary kernel memory overwrites are hard to find this days. But if you stumble on one, you are faced with what is usually known as 'write-what-where' dilemma. What you can write: the byte content and the overwrite size (normally the smaller, the better). Where you can write: a safe kernel address that allows us to intercept the cpu code execution flow within a privileged code segment.
The knowledge of the 'write-what-where' is crucial for a successful exploitation, as is essential for the final phase of a generic exploit: the triggering phase.

As I said before, for the memory overwrite, the smaller, the better. So, if we could just write one bit in kernel address space and that would allow the shellcode to be executed in kernel mode, it would be great. But sometimes, constrains show up, and we can't just write one single bit. The ideal memory address buffer would then be the one it would allow a large spectrum of overwrite widths. This would give it some resilience and stability and would elect it for usage of a broader range of exploits.
As to the 'where' goes, Windows Kernel Patch Protection is a player. Kernel Patch Protection makes it its business to difficult life to those who try to hook on Windows. It is indeed a huge determent.

This document is about a new 'write-what-where' that allows you to easily escalate from user mode if you can write from a DWORD up to a bit, from user mode to kernel memory. This text is not about a vulnerability exploit.

The idea is the following: If you create a GUI application, the GUI application will register a WndProc function with RegisterClassEx for GUI message handling. As soon as the GUI application calls CreateWindow, it asks the kernel (Win32k.sys counterpart) to instantiate the WNDCLASSEX provided by the application upon registration. The kernel then, when needed, lowers its execution level to user mode, and calls back the application registered WndProc. WndProc starts receiving GUI messages and returns it's execution to wait in the kernel for more messages delivery. This is the normal, known usual execution of GUI applications.
The tweak here is that the messaging subsystem kernel counterpart (Win32k.sys) allows for kernel sided registered windows. These windows process their messages in kernel mode, and what differentiates a user mode from a kernel mode window is just a single bit, kept in the third byte of the state field of a tagWND kernel structure at offset 0x14. If you just flip this bit, the WndProc runs in kernel mode as it will be shown. And, if you can't write just a bit, the state field can be overwritten up to its full DWORD size. The bonus here is that all the information we need to get to the critical address is available from user mode, by iterating the aheList. The bit value that defines a window as kernel mode is defined here as:

```
#define WS_EXECUTE_IN_KERNEL        0x04
```

After writing 'Unpack me if you can' where I exposed some details about the Desktop heap, I continued digging on Win32k internals. If you aren't familiar with it, please read it as you need to grasp how the desktop heap is organized to understand some of the steps taken here. I will also give you a small introduction on how the aheList is structured, and how it can be used from user mode, to gather the information we need to find the kernel overwrite spot. I'll leave for a future write a more descriptive text about the aheList.

The aheList is an array of handle entries stored in kernel address space, but mapped read only into process user space memory, when processes register for GUI processing. The aheList comprises all the GUI handles for a Windows session, making this memory area

shared by all GUI processes that run in the same session. The aheList address can be obtained from user mode from the global variable gSharedInfo exported from user32.dll.

```
x user32!gSharedInfo
76dc9440 USER32!gSharedInfo = <no type information>
```

The gSharedInfo is a structure of type win32k!tagSHAREDINFO, as can be seen here:

```
dt win32k!tagSHAREDINFO 76dc9440
   +0x000 psi              : 0x018b0578 tagSERVERINFO
   +0x004 aheList          : 0x017f0000 _HANDLEENTRY
   +0x008 HeEntrySize      : 0xc
   +0x00c pDispInfo        : 0x018b1728 tagDISPLAYINFO
   +0x010 ulSharedDelta    : 0xfdd20000
   +0x014 awmControl       : [31] _WNDMSG
   +0x10c DefWindowMsgs    : _WNDMSG
   +0x114 DefWindowSpecMsgs : _WNDMSG
```

From it, we can get the aheList address, 0x017f0000.

For the purpose of this demonstration I'll be using notepad.exe again. Let's first set a breakpoint on notepad's registered WndProc:

```
bp notepad!NPWndProc
```

As soon as we hit the breakpoint, we get the registers:

```
r
eax=c0000000 ebx=00000000 ecx=00000000 edx=00000003 esi=0000001e
edi=000bfd48
eip=001314de esp=000bfcd0 ebp=000bfcf8 iopl=0         nv up ei pl nz na
pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000206
notepad!NPWndProc:
001b:001314de 8bff            mov     edi,edi
```

We can observe the user mode segments: CS=0x1b, DS=ES=SS=0x23.
The callback declaration format is:

```
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);
```

So, dumping the stack, the first argument will be window handle associated with our thread: 0x0018009e

```
kbL
ChildEBP RetAddr  Args to Child
000bfccc 76d7c4e7 0018009e 0000001e 00000000 notepad!NPWndProc
000bfcf8 76d7c5e7 001314de 0018009e 0000001e
USER32!InternalCallWinProc+0x23
```

Let's go to the aheList and grab the tagWND object, by using 0x0018 as a validator and 0x009e as an index into the aheList array:

```
dt win32k!_handleentry 0x017f0000+@@(sizeof(win32k!_handleentry))*9e
   +0x000 phead            : 0xfe83c9e8 _HEAD
   +0x004 pOwner           : 0xff9d9008 Void
   +0x008 bType            : 0x1 ''
   +0x009 bFlags           : 0 ''
   +0x00a wUniq            : 0x18
```

Let's validate in kernel the information obtained:

```
dt win32k!_head 0xfe83c9e8
   +0x000 h                : 0x0018009e Void
   +0x004 cLockObj         : 9

dt win32k!_THRDESKHEAD 0xfe83c9e8
   +0x000 h                : 0x0018009e Void
   +0x004 cLockObj         : 9
   +0x008 pti              : 0xff9d9008 tagTHREADINFO
   +0x00c rpdesk           : 0x8585d678 tagDESKTOP
   +0x010 pSelf            : 0xfe83c9e8 "???"

dt win32k!tagWND 0xfe83c9e8
   +0x000 head             : _THRDESKHEAD
   +0x014 state            : 0x40020049  <----
   +0x014 bHasMeun         : 0y1
...
   +0x050 rcClient         : tagRECT
   +0x060 lpfnWndProc      : 0x001314de     long  notepad!NPWndProc+0
   +0x064 pcls             : 0xfe80db68 tagCLS
```

Everything looks valid. From user mode, we obtained the corresponding tagWnd address `0xfe83c9e8`. To convert the user mode window to a kernel mode one, we need to set the third bit from the third byte of the state field in the tagWND structure.
Summing the tagWnd address (`0xfe83c9e8`), the state field offset within the tagWnd structure (0x14) and the state offset byte (0x2), we get the kernel memory address that needs to be altered. From here on, we would need an arbitrary memory overwrite to set our value:

```
((BYTE*)&pWnd->state)+0x2 |= WS_EXECUTE_IN_KERNEL
```

Let's fire up the window to a kernel mode window by hand:

```
eb 0xfe83c9e8+0x14+0x2 4
```

Notice that I set the third byte to the `WS_EXECUTE_IN_KERNEL` value because the content is not important as long the flag is set. But for correctness, one could collect the previous window state value (usually 2) from the Desktop heap and OR it with `WS_EXECUTE_IN_KERNEL`. See 'Unpack me if you can'.

Run the machine and we hit the breakpoint again:

```
r
eax=00002cac ebx=00000133 ecx=8a4e3cac edx=8a4e4000 esi=fe83c9e8
edi=ff9d9008
eip=001314de esp=8a4e3c58 ebp=8a4e3c94 iopl=0         nv up ei pl nz na
pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00000206
notepad!NPWndProc:
001314de 8bff            mov     edi,edi
```

Yeh! We just set notepad.exe executing in kernel mode. From now on, the code on WndProc just needs to verify the code segment to validate the kernel mode execution and run the shellcode. To clean up, the shellcode needs just to restore the window state to the previous user mode state, and return from WndProc.

How to solve this? The 'problem' lies in xxxSendMessageTimeout as it does not validate if the WndProc address is in user mode before calling it:


So a simple address validation would suffice to stop this from working with a simple bit flip, but wouldn't stop other kind of approaches.

I'll be waiting for your exploits.