An NCC Group Publication

Exploiting the win32k!xxxEnableWndSBArrows use-after-free (CVE 2015-0057) bug on both 32-bit and 64-bit

Prepared by:
Aaron Adams

tl;dr

Earlier this year I worked on an exploit for an interesting use-after-free vulnerability in win32k.sys (CVE-2015-0057) and was able to develop a reliable exploit on both 32-bit and 64-bit, affecting XP through Windows 8.1 (with a few exceptions). This writeup describes in detail how I approached exploitation on both architectures, which ended up being somewhat different. I also describe how exploitation works on Windows 8.1 with SMEP and in a low integrity environment.

The post is quite long, but I try to provide a lot of detail to demonstrate what is involved in exploiting this bug instead of glazing over details, although I do still glaze over some. Hopefully the level of detail is helpful.

Introduction

On February 10, 2015, Microsoft released MS15-10 to address a number of vulnerabilities. The bug was found by Udi Yavo of enSilo. Udi released a nice analysis of the vulnerability on the breaking malware blog. I recommend reading it to better understand the bug, although I do try to explain most of the details here, as I had to overcome a few hurdles to get it to trigger. This bug was really interesting to exploit, but there are a lot of details omitted from the blog post by Udi, which he acknowledges:

```
Responsible disclosure: although this blog entry is technical, we
won’t reveal any code, or the complete details, to prevent any tech
master from being able to reproduce an exploit.
```

As an added bonus to exploiting this bug we get to evolve into our next Pokémon form: tech wizard. I want to give Udi credit for finding the bug, providing the information he did, and exploiting the bug, which he demonstrates on his blog. It was really helpful.

I had never exploited a win32k.sys vulnerability before, and was not familiar with usermode callbacks or many of the APIs I was using, so I would also like acknowledge the amazing resources made available online by a few well-known security researchers: Skywing, Tarjei Mandt, Alex Ionescu, j00ru, etc. All of these people deserve massive kudos for providing so much technical information publicly. One of the papers I used extensively was Tarjei Mandt’s Win32k.sys exploitation paper. I highly recommend reading that if you have no familiarity with win32k.sys.

Since I wrote my exploit, a nice reverse-engineered exploit was made available for CVE-2015-1701, which is useful for seeing an actual code example of how to hook usermode callbacks. Kudos to those who reversed that and made it available.

It's probably worth noting that most of my analysis below was done on a Windows 7 installation, because it appears to be the only version that has corresponding symbols available for most of the win32k.sys structures. Microsoft pulled the information out again as of 8, for an unknown reason.

Lastly I want to say that the way I approach exploiting this bug is quite complicated. It's entirely possible that there is a much easier way to do it that I just overlooked. I'd love to hear if someone did it a different way. Either way I hope what's described is still useful for people researching win32k.sys bugs.

The bug

The following shows the, fairly subtle, bug when you look at it just in the disassembly of `win32k!xxxEnableWndSBArrows`:

Unpatched

```
.text:FFFFF97FFF1B157D                 mov     r8d, r13d
.text:FFFFF97FFF1B1580                 mov     rdx, r14
.text:FFFFF97FFF1B1583                 call    xxxDrawScrollBar
; xxxDrawScrollBar could've issued usermode callback
.text:FFFFF97FFF1B1588                 jmp     short
loc_FFFFF97FFF1B1519
[...]
.text:FFFFF97FFF1B1519                 mov     eax, [rbx]
; Dereference tagSBINFO ptr without sanity check
.text:FFFFF97FFF1B151B                 mov     ebp, 0FFFFFFFBh
.text:FFFFF97FFF1B1520                 xor     eax, esi
```

In the code above, `win32k!xxxDrawScrollBar` can under the right circumstances make a call to userland at which point the `tagSBINFO` can be freed by the attacker. When returning to the code above, the instruction at 0xFFFFF97FFF1B1519 will then use the now invalid pointer.

Patched

```
.text:FFFFF97FFF1D69C3                 xor     r8d, r8d
.text:FFFFF97FFF1D69C6                 mov     rdx, rbp
.text:FFFFF97FFF1D69C9                 call    xxxDrawScrollBar
; Could've issued usermode callback
```

```
.text:FFFFF97FFF1D69CE                cmp     rbx, [rdi+0B0h]
; Is tagSBINFO ptr still correct?
.text:FFFFF97FFF1D69D5                jz      short
loc_FFFFF97FFF1D69E4 ; If so, continue
.text:FFFFF97FFF1D69D7
.text:FFFFF97FFF1D69D7 loc_FFFFF97FFF1D69D7:
.text:FFFFF97FFF1D69D7                mov     rcx, rbp
.text:FFFFF97FFF1D69DA                call    _ReleaseDC
.text:FFFFF97FFF1D69DF                jmp     loc_FFFFF97FFF1D6958
; Jump to exit function block
.text:FFFFF97FFF1D69E4 ; ---------------------------------------------
------------------------------
.text:FFFFF97FFF1D69E4
.text:FFFFF97FFF1D69E4 loc_FFFFF97FFF1D69E4:
.text:FFFFF97FFF1D69E4                mov     eax, [rbx]
; Safely deref the unchanged tagSBINFO ptr
.text:FFFFF97FFF1D69E6                xor     eax, r14d
```

In the patched version above, we see that the original `tagSBINFO` pointer is checked for NULL before being dereferenced. More information about the related structures will be provided later.

The Basics - Getting to stage 1 memory corruption

When building the exploit, we achieve corruption in a series of stages. So we can think of triggering the bug as stage one corruption.

Technically the root of the issue is a use-after-free on the desktop heap (more on that heap later). This was confusing to me at first, because I wasn't familiar with usermode callback functions used by win32k.sys or how they were expected to work, so I thought the problem might actually be a race condition due to some locking issue that in turn let you trigger the use-after-free. But in the end, the locking of the structures that actually support locking was correct and the way things are behaving on that front is expected. The real problem in a nutshell is:

1. The `win32k!xxxEnableWndSBArrows` function holds a desktop heap pointer to a `tagSBINFO` struct, used to describe a scrollbar, that it reads out of the associated window's `tagWND` struct.
2. The `win32k!xxxEnableWndSBArrows` function makes a call to a function that can result in a usermode callback (that can be hooked).

3. Once code is executing in userland, changes to structures on the desktop heap can occur by calling other win32k.sys system calls, including freeing the `tagSBINFO` struct from the desktop heap.
4. Upon returning to kernel mode, the `win32k!xxxEnableWndSBArrows` does not re-reference and validate the original `tagSBINFO` pointer from the `tagWND` struct it was originally obtained from (which would now indicate it as being freed), but instead keeps using the now-stale pointer.

That's it. Ignoring how usermode callbacks work for now, this part is pretty straightforward.

Understanding what we control

But what does this actually let us corrupt and why? As Udi's blog post mentions, you can effectively set or unset two bits from what the code believes is the `WSBFlags` member of the `tagSBINFO` structure. This isn't really an ideal use-after-free scenario, but the paper gives a hint how to leverage this, which I will describe in the next section. But first, let's better understand how we can control the bit manipulation.

First let's look at the `tagSBINFO` structure (consistent across 32/64-bit):

```
kd> dt -b !tagSBINFO
win32k!tagSBINFO
   +0x000 WSBflags         : Int4B
   +0x004 Horz             : tagSBDATA
      +0x000 posMin           : Int4B
      +0x004 posMax           : Int4B
      +0x008 page             : Int4B
      +0x00c pos              : Int4B
   +0x014 Vert             : tagSBDATA
      +0x000 posMin           : Int4B
      +0x004 posMax           : Int4B
      +0x008 page             : Int4B
      +0x00c pos              : Int4B
```

The UAF bug lives in `win32k!xxxEnableWndSBArrows(),` which is responsible for enabling and disabling one or both of the horizontal and vertical scrollbar arrows associated with a scrollbar control. A scrollbar control is effectively a special window used to manipulate a scrollbar. It can be created with `CreateWindow()` by using the built-in `"SCROLLBAR"` system class.

The function prototype for `win32k!xxxEnableWndSBArrows()` is:

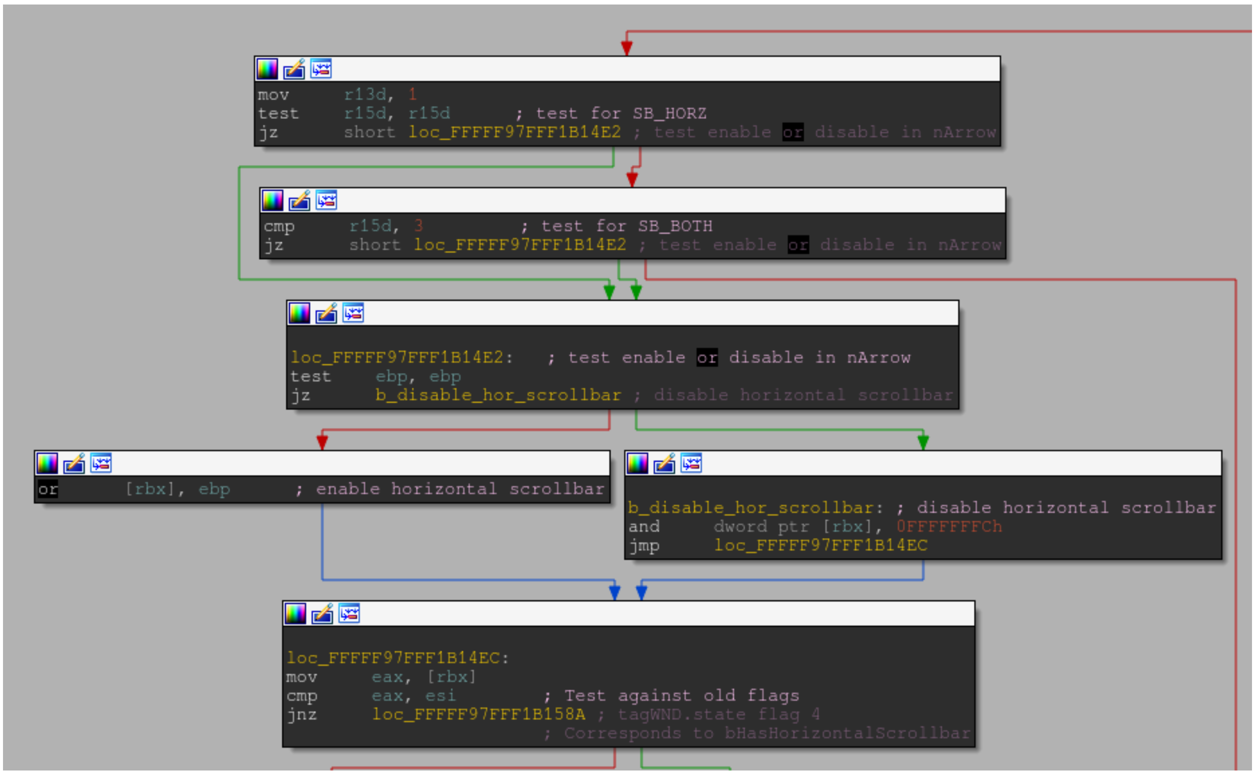
```
BOOL xxxEnableWndSBArrows(PWND wnd, UINT WSBflags, UINT wArrows);
```

The `WSBFlags` parameter corresponds to the scrollbar userland constants in `WinUser.h`, and specifies which scrollbar will actually be operated on:

```
#define SB_HORZ             0
#define SB_VERT             1
#define SB_CTL              2
#define SB_BOTH             3
```

The `wArrows` parameter actually specifies the state of the arrows, which is either enabled or disabled. Bits being set mean the arrow is disabled and bits unset mean the arrow is enabled. The least significant two bits of `wArrows` correspond to the horizontal scrollbar. The next two bits correspond to the vertical scrollbar. The rest of the `wArrows` bits don't matter for the sake of exploitation.

The following code, from `win32k!xxxEnableWndSBArrows()`, shows setting or unsetting the horizontal arrow bits, if the `SB_HORZ` or `SB_BOTH` flag was set:



```
mov     r13d, 1
test    r15d, r15d       ; test for SB_HORZ
jz      short loc_FFFFF97FFF1B14E2 ; test enable or disable in nArrow
```

```
cmp     r15d, 3          ; test for SB_BOTH
jz      short loc_FFFFF97FFF1B14E2 ; test enable or disable in nArrow
```

```
loc_FFFFF97FFF1B14E2:   ; test enable or disable in nArrow
test    ebp, ebp
jz      b_disable_hor_scrollbar ; disable horizontal scrollbar
```

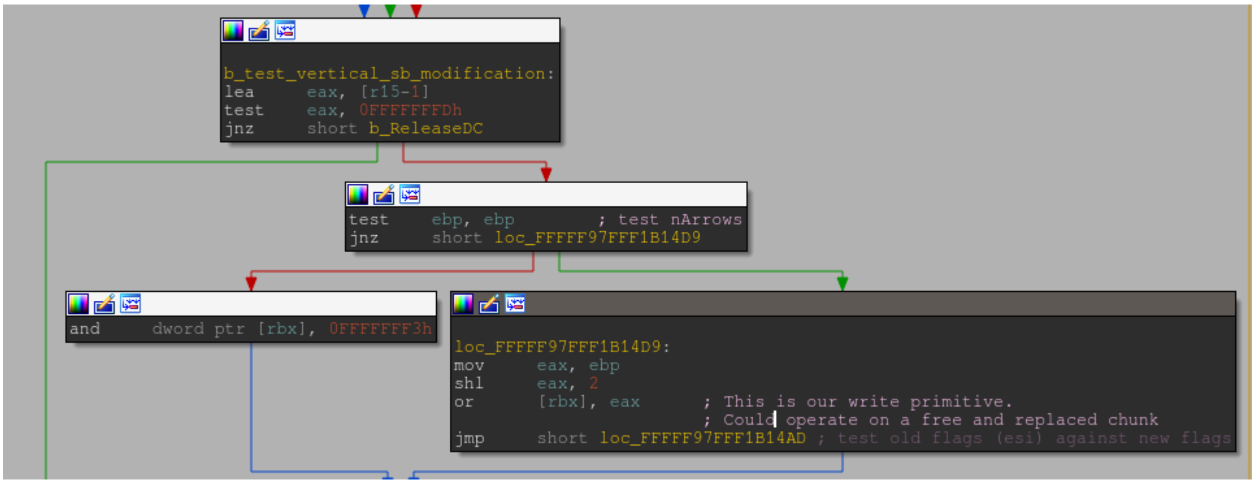
```
or      [rbx], ebp       ; enable horizontal scrollbar
```

```
b_disable_hor_scrollbar: ; disable horizontal scrollbar
and     dword ptr [rbx], 0FFFFFFFCh
jmp     loc_FFFFF97FFF1B14EC
```

```
loc_FFFFF97FFF1B14EC:
mov     eax, [rbx]
cmp     eax, esi         ; Test against old flags
jnz     loc_FFFFF97FFF1B158A ; tagWND.state flag 4
                         ; Corresponds to bHasHorizontalScrollbar
```

The bug actually manifests between setting the horizontal and vertical scrollbar flags. After updating the horizontal scrollbar, as long as the window associated with the scrollbar is currently visible on the desktop, the `win32k!xxxEnableWndSBArrows()` function will call `win32k!xxxDrawScrollBar()`, which the original paper notes can potentially drive down into a usermode callback.

Before we discuss usermode callbacks, let's first continue to discuss what happens after the call to `win32k!xxxDrawScrollBar()`. This is effectively the same logic as for the horizontal bar, but slightly different bits. If we chose to disable the vertical scrollbar, and assuming we triggered the use-after-free, this will write two bits into whatever is now allocated in place of the `tagSBINFO` chunk. So assuming the value was originally 0x2, it would now be 0xe. This is shown in the figure below.



```
b_test_vertical_sb_modification:
lea     eax, [r15-1]
test    eax, 0FFFFFFFDh
jnz     short b_ReleaseDC
```

```
test    ebp, ebp            ; test nArrows
jnz     short loc_FFFF97FFF1B14D9
```

```
and     dword ptr [rbx], 0FFFFFFF3h
```

```
loc_FFFF97FFF1B14D9:
mov     eax, ebp
shl     eax, 2
or      [rbx], eax          ; This is our write primitive.
                            ; Could operate on a free and replaced chunk
jmp     short loc_FFFF97FFF1B14AD ; test old flags (esi) against new flags
```

This bit flip is enough to eventually get code execution. I did not investigate a way to achieve exploitation by unsetting bits, but it might be possible.

One important thing about the above is that in order for both the horizontal and vertical bars to actually be operated on, the scrollbar must have been created in a way that indicates it has both. This involves setting the `WS_HSCROLL` and `WS_VSCROLL` flags when calling `CreateWindow()`. An example is below:

```
        g_hSBCtl = CreateWindowEx(
                0,                                  // No extended style
                "SCROLLBAR",                        // class
                NULL,                              // name
                SBS_HORZ | WS_HSCROLL | WS_VSCROLL, // need both SB
types
                10,                                // x
                10,                                // y
                100,                               // width
                100,                               // height
                g_hSpray[UAFWND],                   // a non control
parent window is required
                (HMENU)NULL,
                NULL,                               // window owner
                NULL                                // extra params
                );
```

You also want to ensure it's visible (it should be by default, but just in case):

```
        result = ShowWindow(g_hSBCtl, SW_SHOW);
```

By default the scrollbars are enabled, once we're ready to try to hit the vulnerable code shown above we can disable them, to eventually set the bits we want to corrupt:

```
        result = EnableScrollBar(g_hSBCtl, SB_CTL | SB_BOTH,
ESB_DISABLE_BOTH);
```

Triggering the bug

So above I explained what the bug is and how to trigger some of the related code, but we're still missing the very important step of intercepting the usermode callback triggered by `win32k!xxxDrawScrollBar()` so we can change the contents of the heap before `win32k!xxxEnableWndSBArrows()` continues running. We need to actually trigger the bug, which, if you don't know anything about `win32k.sys` or any of these APIs, as was the case with me starting out, is an adventure on its own.

The original paper contains a good call-stack diagram showing that deep within the functionality triggered by the`win32k!xxxDrawScrollBar()` call, the `ClientLoadLibrary()` function will be called, and gets dispatched through the `KeUserModeCallback()` function. We need to figure out what exactly `KeUserModeCallback()` calls, so we can try to hook it in our process.

I found a few good papers that had bits and pieces about how usermode callbacks work. The posts and papers that touch on it, amongst other win32k areas that I found very useful, are:

- https://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf (Tarjei's white paper)
- http://azimuthsecurity.com/resources/recon2012_mandt.pptx (Tarjei's slides with extra info)
- http://www.nynaeve.net/?p=204
- http://www.cprogramdevelop.com/3825874/

- http://www.zer0mem.sk/?p=410
- https://www.reactos.org/wiki/Techwiki:RegisterUserApiHook
- http://pasotech.altervista.org/windows_internals/Win32KSYS.pdf
- http://j00ru.vexillium.org/?p=614
- http://uninformed.org/index.cgi?v=10&a=2#SECTION00042000000000000000

Basically, each process maintains a table of usermode callback function pointers, to which the `PEB->KernelCallBackTable` member points. When the kernel wants to call a usermode function it will pass a function index to `KeUserModeCallBack()`. In the case above, the index corresponds to the `__ClientLoadLibrary()` function in userland.

`KeUserModeCallBack()` will end up calling the `KiUserModeCallbackDispatch()` function in userland, which in turn looks up the index in the `PEB->KernelCallBackTable` and executes it.

In order to hook a given entry, you can look up the `PEB->KernelCallBackTable` table and patch out the `__ClientLoadLibrary()` index directly. It should be noted that these indices can differ for each OS version, but are consistent across architectures.

If we want to investigate the `PEB->KernelCallBackTable` table to see what's in there and to work out indices, we find the address of the table using WinDbg. Note that I alternate between 32-bit and 64-bit for examples where it shouldn't make a big difference:

```
kd> dt !_PEB @$peb
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0 ''
   +0x003 BitField         : 0x8 ''
   +0x003 ImageUsesLargePages : 0y0
[...]
   +0x02c KernelCallbackTable : 0x76daf620 Void

kd> dds 0x76daf620
76daf620  76d96443 user32!__fnCOPYDATA
76daf624  76ddf0e4 user32!__fnCOPYGLOBALDATA
76daf628  76da736b user32!__fnDWORD
76daf62c  76d9d603 user32!__fnNCDESTROY
76daf630  76dc50f9 user32!__fnDWORDOPTINLPMSG
76daf634  76ddf1be user32!__fnINOUTDRAG
76daf638  76dc6cd0 user32!__fnGETTEXTLENGTHS
76daf63c  76ddf412 user32!__fnINCNTOUTSTRING
```

```
76daf640  76d9ce49 user32!__fnINCNTOUTSTRINGNULL
[...]
76daf724  76da3962 user32!__ClientLoadLibrary

kd> ?? (0x76daf724-0x76daf620)/4
int 0n65
```

In the example above we know that the `__ClientLoadLibrary` function is index 65, so that is the entry we want to hook. What I noticed after hooking is that the `__ClientLoadLibrary` function is called a lot by win32k-related code! The first thing I needed to do was indicate to my hook right before I actually triggered the call that we're interested in, so that we could know exactly which call into the hook we needed to change. So the hook code checks a global flag, and only tries to do something interesting if it's set.

There were then two more hurdles:

1) If I let the original `__ClientLoadLibrary` functions behave normally, when I actually triggered the vulnerable call in win32k.sys I found that it never ended up actually making its way into userland. I didn't investigate this too heavily, but I assume it's possibly because whatever library it's loading for this call is already loaded so it determines it doesn't need to call the function again. In order to work around this I had my hook manipulate every call into `__ClientLoadLibrary` to return no result, which forces it to retry to load the library constantly. I worked out that passing back NULLs in the structure parameter was enough just by reversing `__ClientLoadLibrary()` in user32.dll.

2) The call into `EnableScrollBar()` ends up triggering `__ClientLoadLibrary` calls before the one that's triggered by `win32k!xxxDrawScrollBar()` which we want to abuse, so I had to work out the number of calls before the one I'm interested in and use a counter so I know to trigger the bug on the exact right call into the hook. Fortunately, this count is stable across both architectures and OS versions.

So the hook looks like this:

```
void
ClientLoadLibraryHook(void * p)
{
        CHAR Buf[PGSZ];

        memset(Buf, 0, sizeof(Buf));
        if (g_PwnFlag) {
                dprintf("[+] __ClientLoadLibrary hook called\n");
                if (++g_HookCount == 2) {
                        g_PwnFlag = 0; // Only fire once..
                        ReplaceScrollBarChunk(NULL);
                }
        }
        fpClientLoadLibrary(&Buf); // call original
}
```

Once we know for sure we've been called specifically from the call to `win32k!xxxDrawScrollBar()` we can try to trigger the bug. For now, since we're just worried about triggering, we can just call `DestroyWindow(g_hSBCtl)`. This will be enough to free the `tagSBINFO` structure from the window The window structure itself won't be freed yet because there is still a reference count as it is still in use by the original call, but the `tagSBINFO` has no such reference-counting mechanism so is freed in the process.

At this point we've triggered the bug. Even if we don't reallocate the now-free chunk that was holding `tagSBINFO`, we will write the two disable bits to whatever is now in that freed heap location. The next step is to replace this freed chunk with one we want to put there, so we can do something more interesting than just flip a couple of bits. In order to do this, we need a bit of background on the desktop heap.

The desktop heap

The desktop heap is used by win32k.sys to store GUI objects associated with a given desktop. This includes window objects and their associated structures, like property lists, window text, and scrollbars. The Tarjei paper touches on this, but what's most important to note is that it's actually just a simplified version of the userland backend allocator that operates using `RtlAllocateHeap()` and `RtlHeapFree()`. The heap is tracked by a `_HEAP` structure like you'd expect. There is no frontend allocator, so no Low Fragmentation Heap, no Lookaside list, etc.

Every time you create a desktop, a heap is created to service it. This means that we can actually allocate a new desktop in order to get a much "fresher" heap that we can more predictably manipulate. However, it is worth noting that a process running in a low-integrity environment is actually not allowed to create a new desktop.

What's primarily of interest for now (we'll cover more details later about the metadata and such) is tracking allocations.

Monitoring desktop heap allocations

To monitor allocations and frees from the desktop heap I used the following WinDbg script:

64-bit heap monitoring

```
ba e 1 nt!RtlFreeHeap ".printf\"RtlFreeHeap(%p, 0x%x, %p)\", @rcx,
@edx, @r8; .echo ; gc";
ba e 1 nt!RtlAllocateHeap "r @$t2 = @r8; r @$t3 = @rcx; gu; .printf
\"RtlAllocateHeap(%p, 0x%x):\", @$t3, @$t2; r @rax; gc";
```

32-bit heap monitoring

```
ba e 1 nt!RtlAllocateHeap "r @$t2 = poi(@esp+c); r @$t3 = poi(@esp+4);
gu; .printf \"RtlAllocateHeap(%p, 0x%x):\", @$t3, @$t2; r @eax; gc";
ba e 1 nt!RtlFreeHeap ".printf\"RtlFreeHeap(%p, 0x%x, %p)\",
poi(@esp+4), poi(@esp+8), poi(@esp+c); .echo ; gc"
```

In addition to these breakpoint scripts, because the desktop heap is actually just a simplified form of the userland backend allocator, we can actually leverage the `!heap` command in WinDBG itself.

Filling heap holes

In order to exploit the bug we need to replace this recently freed `tagSBINFO` chunk, but we also know from how these bugs are typically exploited that we'll eventually be corrupting some adjacent data. This gives us the fundamental requirement of predictably allocating chunks of interest adjacent to our corrupted structure. And in order to predict where a chunk is allocated, we must be in control of the entire heap layout (or as much as possible). The logical way to go about this is to try to fill in as many free chunks as we can so all new allocations are adjacent and if we need holes we can create them at predictable locations (by freeing the associated chunk).

Part of this is simply understanding side effect allocations, which the WinDbg scripts above can help with. Tarjei mentioned most of the main objects of interest allocated on this heap in his win32k slides, which I found to be pretty consistent with what I was seeing. His list is:

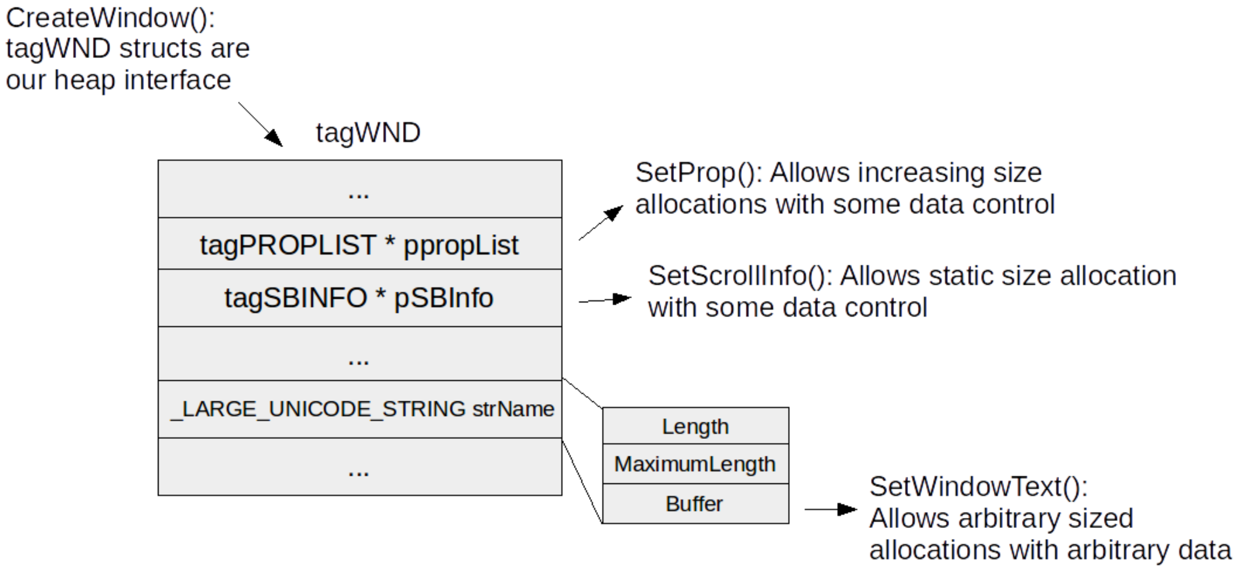
- Window
- Menu
- Hook
- CallProcData
- Input Context

The desktop heap is pretty interesting in that most allocations are directly tied to window objects, managed by the `tagWND` structure, which means if we want to allocate a chunk of an arbitrary size (say a small size to fill a small hole), then we first need an allocated window to interact with. You can basically think of a window structure as an allocation interface to the heap. Another point of interest is that many allocations you can create through a window cannot subsequently be destroyed without destroying the window itself, which obviously has heap side effects. Lastly, let's assume a window allows you to allocate a chunk of size N. Just for the sake of example, what if, for whatever reason, we need twenty allocations of size N? The actual things in a window that let us allocate an arbitrary size are not stored on lists. So each window lets you do one controlled allocate of size N. So if you need to make twenty allocations of size N, you must first create twenty windows and use each window to facilitate the allocation.

There are three additional important datatypes, also allocated on this heap, that we can indirectly use for controlling data on the heap via window objects. I will use these pretty extensively for exploitation and feng shui. These are:

1. `tagPROPLIST` structures: these serve as a small enough allocation that they will fill any small holes we're worried about. A window containing a single `tagPROPLIST` entry will allocate 0x10 bytes on 32-bit and 0x18 bytes on 64-bit.
2. Window text: This is an arbitrary sized UNICODE string allocation on the desktop heap, which is stored in a `_LARGE_UNICODE_STRING` structure embedded in the `tagWND` structure. Note that the `strName` member is a structure, not a pointer, but the structure will contain a pointer to an associated window text allocation.
3. `tagSBINFO` structures: the source of the vulnerability, but also contain four partially or fully controlled members.

The following diagram demonstrates the relationship between these datatypes:



CreateWindow():
tagWND structs are
our heap interface
tagWND
...
tagPROPLIST * ppropList
tagSBINFO * pSBInfo
...
_LARGE_UNICODE_STRING strName
...
SetProp(): Allows increasing size allocations with some data control
SetScrollInfo(): Allows static size allocation with some data control
Length
MaximumLength
Buffer
SetWindowText():
Allows arbitrary sized
allocations with arbitrary data

To do the initial heap filling I create a large number of `tagWND` structures (by creating new Windows). This has the effect of filling a lot of big holes on the heap, and also gives us interfaces in order to do other allocations as needed. On Windows 8 and Windows 8.1 allocating a new window results in an auto-allocation of a `tagPROPLIST` structure (which you can observe during development using the WinDbg scripts above). On Windows 7 and earlier we allocate new `tagPROPLIST` entries ourselves, and these serve to fill any small holes.

At this point every Window we sprayed has no corresponding window text strings value, so we could still use those for arbitrary size allocations and frees as needed. Once created you can't actually remove an existing property list without destroying the corresponding Window, but you can force the list to be reallocated to accommodate a new entry, which can be used to create a hole at the previous location. To do this you simply need to set a new property with an identifier (`atomKey`) that doesn't already exist in the list.

Validating feng shui layouts

Interestingly, the desktop heap is mapped as read-only into userland. This means that we can validate the feng shui layouts we're trying to create and bail out if things didn't work out. First we need to figure out where the userland map of the desktop heap is. This is once again described by Tarjei in his win32k paper. The PEB contains an undocumented structure called `Win32ClientInfo`, which is defined approximately as follows:

```
typedef struct _CLIENTINFO
{
        ULONG_PTR CI_flags;
        ULONG_PTR cSpins;
        DWORD dwExpWinVer;
        DWORD dwCompatFlags;
        DWORD dwCompatFlags2;
        DWORD dwTIFlags;
        PDESKTOPINFO pDeskInfo;
        ULONG_PTR ulClientDelta;
        // incomplete. see reactos
} CLIENTINFO, *PCLIENTINFO;
```

First the `PDESKTOPINFO` structure contains the following:

```
typedef struct _DESKTOPINFO { PVOID pvDesktopBase; PVOID pvDesktopLimit;
// incomplete. see reactos } DESKTOPINFO, *PDESKTOPINFO;
```

The `pvDesktopBase` contains the kernel address of the desktop heap, which we record. Next the `ulClientDelta`value from the `Win32ClientInfo` structure contains a delta which is the offset between the userland mapping and the kernel mapping, which tells us the information.

However, we don't want to have to parse the heap ourselves if we don't have to, so we ideally also want to be able to take a given user32 handle, like an `HWND` value, and convert it to the address in the userland mapping, so we can actually determine where it is in relation to other allocations. In order to do `HANDLE` lookups, we need to find a structure called `gShared`, which is normally stored in `user32.dll`. On Windows 7 and later this address is exported, so is easy to find.

On most systems the structure is defined as follows:

```
kd> dt !tagSHAREDINFO
win32k!tagSHAREDINFO
   +0x000 psi              : Ptr32 tagSERVERINFO
   +0x004 aheList          : Ptr32 _HANDLEENTRY
   +0x008 HeEntrySize      : Uint4B
   +0x00c pDispInfo        : Ptr32 tagDISPLAYINFO
   +0x010 ulSharedDelta    : Uint4B
   +0x014 awmControl       : [31] _WNDMSG
   +0x10c DefWindowMsgs    : _WNDMSG
   +0x114 DefWindowSpecMsgs : _WNDMSG
```

In the structure above, `aheList` is a pointer to an array of handles, and each `_HANDLEENTRY` contains a pointer to the actual kernel address of the handle. We can then subtract our known userland delta from it and have a usable address to investigate.

Unfortunately, finding the `gSharedInfo` data on systems earlier than Windows 7 is not so easy, as the symbol is not exported. Tarjei's paper states that the undocumented `CsrClientConnectToServer` function could be used to obtain a copy of `gSharedInfo`, but I could find no working examples. One annoying hurdle with implementation is that the size of the structures needed by the function change between 64-bit Vista, 32-bit Vista, and 64-bit and 32-bit Windows XP, so you can’t quite trust what you find in ReactOS from my experience.

Once we figure out where things are mapped, we can build functions that tell us exactly where window objects are on the desktop heap. Then if we want to know where a corresponding property list or text chunk was allocated, we can just parse that structure at that location in userland.

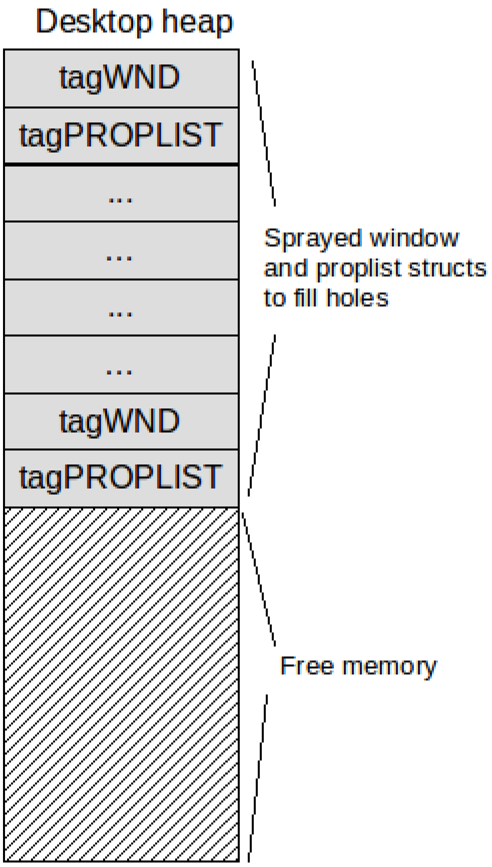
Replacing tagSBINFO with tagPROPLIST

Now we're finally getting closer to exploiting this. We have a way to massage the heap, a way to validate that our chunks are in the right positions, and we can trigger the bug, so now we can finally ensure the freed `tagSBINFO` chunk is replaced with a `tagPROPLIST` property list of our choosing. Note that because `tagPROPLIST` is just the header of a larger list, we are able to match the size of the list to the scrollbar info chunk, which we'll describe shortly. It is basically an array of `tagPROP` entries, but is called a property list; so I will use the terms array and list interchangeably. A `tagPROPLIST` structure looks like the following on 64-bit:

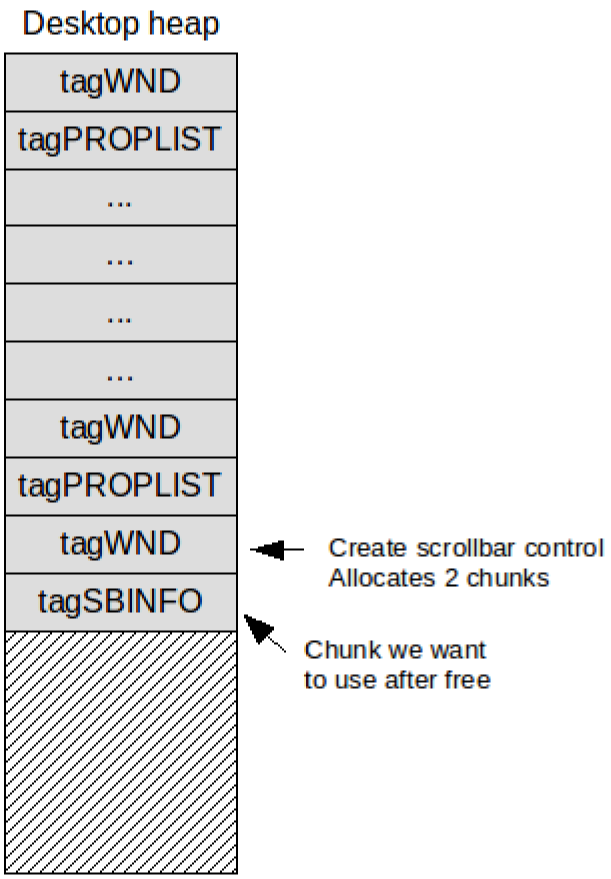
```
kd> dt -b !tagPROPLIST
win32k!tagPROPLIST
   +0x000 cEntries         : Uint4B
   +0x004 iFirstFree       : Uint4B
   +0x008 aprop            : tagPROP
      +0x000 hData            : Ptr64
      +0x008 atomKey          : Uint2B
      +0x00a fs               : Uint2B
```

As alluded to earlier, a property list is associated with a window. Property lists are created with the `SetProp()` function. It works by searching for an existing property with a matching `atomKey,` and if one isn't found, a new property entry is created within the property list. If no property list is found at all, one is allocated and linked into the `tagWND` structure.

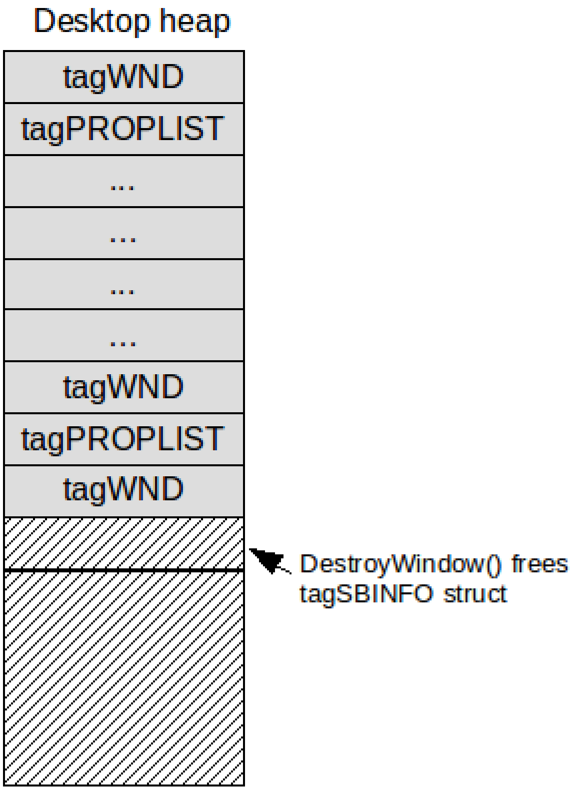
So assuming we've sprayed a bug of `tagWND` structures and created associated `tagPROPLIST` entries for each one, this ends up with a layout similar to that shown below:



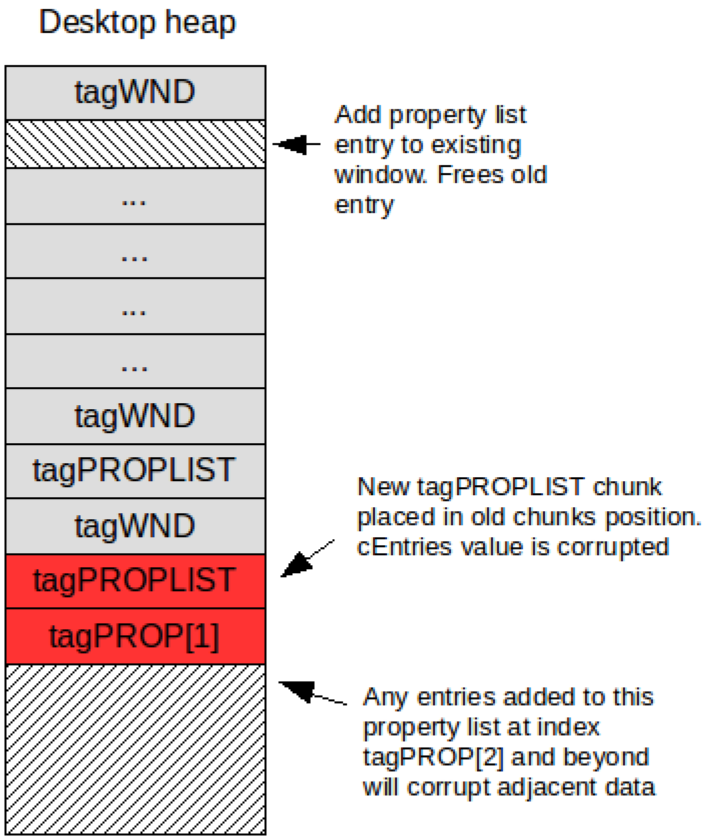
Once this is set up, we can allocate the scrollbar control we want to abuse. This will result in something similar to the following:



Then we interact with the scrollbar control, causing our hooked usermode callback to fire, which lets us free the `tagSBINFO` structure by attempting to destroy the window. This results in a layout similar to the following:



On 64-bit, a `tagSBINFO` structure is 0x28 bytes and a single entry `tagPROPLIST` array is 0x18 bytes, 0x10 bytes of which are the default `tagPROP` entry. So a property list with two entries will be 0x28 bytes (0x8 + 0x10 + 0x10), which is a perfect fit. Let's assume we've sprayed memory so that we have all holes filled. We'll just need to use one window with a pre-existing property list and plan to add a new entry to its list immediately after freeing the `tagSBINFO` (already illustrated in previous diagrams). What this does is free the 0x18 chunk associated with the original `tagPROPLIST` structure, which due to heap spraying won't be adjacent to free chunks and therefore won't coalesce into anything to create a chunk large enough to hold the subsequent 0x28 byte allocation we're hoping for. Instead the recently freed `tagSBINFO` location will be used. This scenario is illustrated below:



Add property list entry to existing window. Frees old entry

New tagPROPLIST chunk placed in old chunks position. cEntries value is corrupted

Any entries added to this property list at index tagPROP[2] and beyond will corrupt adjacent data

When we return from the callback we hooked, the UAF will trigger and bits will be written into the `tagPROPLIST`, specifically the `cEntries` member. Originally `cEntries` is 0x2, corresponding to the two property list entries we've created. After corruption it becomes 0xe, corresponding to bits three and four (counting from one) being set.

From this point we've created a new memory corruption primitive. Any time we add a property list to this corrupted `tagPROPLIST` entry, up to 0xc more entries, we will overwrite whatever is adjacent on the heap. I refer to this as stage two corruption.

Property list abuse - stage two corruption

In Udi’s blog this is really as far as the explanation went. This was described as a "traditional buffer overflow" from this point onwards, however in my experience it was still very difficult to go from this point to an arbitrary read/write primitive or a way to get code execution. Let's revisit the `tagPROPLIST` structure on 64-bit:

```
kd> dt -b !tagPROPLIST
win32k!tagPROPLIST
   +0x000 cEntries          : Uint4B
   +0x004 iFirstFree        : Uint4B
```

```
    +0x008 aprop            : tagPROP
       +0x000 hData            : Ptr64
       +0x008 atomKey          : Uint2B
       +0x00a fs               : Uint2B
```

Stage one has left us with a corrupted `tagPROPLIST` array that allows us to write additional `tagPROP` structures. There are only two members in a `tagPROPLIST` :

- `cEntries`: Indicates the total number of entries the list can hold.
- `iFirstFree`: Indicates the index of the first free entry. A full list (meaning a new one should be allocated) is indicated by `iFirstFree == cEntries`.

When a new property entry is being inserted into a list, a function is first called to scan every entry up until the `iFirstFree` index is hit. At no point is there a check to see if `iFirstFree` is greater than `cEntries` during this logic. If the corresponding `atomKey` isn't found in the list, then a check is done to ensure that `iFirstFree != cEntries`. If true then a new entry is inserted at index `iFirstFree`. If the size test was false (meaning `iFirstFree == cEntries`) then a new list that includes space for the new entry will be allocated, at which point the list is copied over and the new entry is added.

The `tagPROP` structures are associated with the `SetProp()` function. The `hData` member corresponds to the `HANDLE hData` argument of `SetProp()`. It is a process-specific value that is simply identified by the `atomKey` member. Fortunately for us, because it is process-specific we can provide any value we want - it is simply opaque data as far as the kernel is concerned. On 64-bit this gives us eight bytes of control, and on 32-bit four bytes.

The `atomKey` member corresponds to the `LPCTSTR lpString` argument. As per the MSDN documentation for `SetProp()`, the caller can pass in either a pointer to a string or a 16-bit atom value. In the case of a string being passed in, this will be automatically converted to an atom prior to actually being stored in the property list. Because we can effectively pass any atom value into `SetProp(),` this gives us the ability to control these two bytes as well; however, there are some constraints. If we're adding a new property list to an array, the unique identifier of the entry is the `atomKey`. Thus when corrupting data we can never repeat the same `atomKey`, otherwise when setting a new entry, it will replace the old entry with the matching key. Finally, the `fs` member is not controlled by us and is set to 0 for corresponding `atomKey` values < 0xBFFF, which correspond to integer atoms.
The `fs` member is set to 2 for `atomKey` values >= 0xC000.

One more point to note is that you might have noticed that the `tagPROP` is only 0xc bytes. This structure ends up being aligned to 0x10 bytes on 64-bit, so we also end up with an additional four bytes that aren't written and so can't be corrupted when inserting a `tagPROP` entry. The last important point is that the start of a chunk holding a `tagPROPLIST` list begins with the 0x8 bytes of data defining the array size, meaning that each new `tagPROP` entry will always be written to an address ending with 0x8.

So for every `tagPROP` we insert, on 64-bit, this gives us:

```
* Offset 0x0: 8 bytes of arbitrary controlled data (hData)
* Offset 0x8: 2 bytes of almost controlled data (atomKey)
* Offset 0xa: 2 bytes of non-controlled data (fs)
* Offset 0xc: 4 bytes of non-modified data (padding)
```

This is far better than two bits, but it's still not great. Unless we can overwrite something with the first eight bytes of data from the fully controlled `hData` member, we'll be quite limited. If we need to write to some member deep within an adjacent structure, we also can't really avoid uncontrolled corruption of certain values. In spending quite a bit of time looking at various objects on the desktop heap, with the above corruption constraints in mind, the only way I could think of to leverage this to build an arbitrary read/write primitive was to corrupt the `strName` member of an adjacent `tagWND` structure, which is a structure of the type `_LARGE_UNICODE_STRING`:

```
kd> dt !_LARGE_UNICODE_STRING
win32k!_LARGE_UNICODE_STRING
   +0x000 Length           : Uint4B
   +0x004 MaximumLength    : Pos 0, 31 Bits
   +0x004 bAnsi            : Pos 31, 1 Bit
   +0x008 Buffer           : Ptr64 Uint2B
```

If we could corrupt the `Buffer` member of this structure we could then operate on the window string to read and write up to `MaximumLength` bytes from the given address. So this is what I did. You might also recognize this structure from the earlier section on how to create chunks on the desktop heap of arbitrary size and data, as it is the exact same thing that can be used there.

Now that we understand how we can use `tagPROPLIST` entries to corrupt data and what parts we control, and most importantly what constraints we are faced with, this is where the techniques for 32-bit and 64-bit diverge. What I did first on 64-bit ended up not working on 32-bit.

What I do next is quickly turn stage two corruption (aka writing with `tagPROP` structs) into yet another corruption primitive that lets us write fully controlled data, which I refer to as stage three corruption.

Building a read/write primitive - Stage three corruption

64-bit

The plan is to corrupt the `strName` member of an adjacent `tagWND` struct. We already know it's a `_LARGE_UNICODE_STRING`, but let's take a look at the `tagWND` structure in more detail to see what this thing we want to target actually looks like:

```
kd> dt !tagWND
win32k!tagWND
   +0x000 head             : _THRDESKHEAD
   +0x028 state            : Uint4B
   +0x028 bHasMeun         : Pos 0, 1 Bit
   +0x028 bHasVerticalScrollbar : Pos 1, 1 Bit
   +0x028 bHasHorizontalScrollbar : Pos 2, 1 Bit
[SNIPPED FLAGS]
   +0x028 bDestroyed       : Pos 31, 1 Bit
   +0x02c state2           : Uint4B
[SNIPPED FLAGS]
   +0x02c bWMCreateMsgProcessed : Pos 31, 1 Bit
   +0x030 ExStyle          : Uint4B
   +0x030 bWS_EX_DLGMODALFRAME : Pos 0, 1 Bit
   +0x030 bUnused1         : Pos 1, 1 Bit
   +0x030 bWS_EX_NOPARENTNOTIFY : Pos 2, 1 Bit
[SNIPPED FLAGS]
   +0x030 bUIStateFocusRectHidden : Pos 31, 1 Bit
   +0x034 style            : Uint4B
   +0x034 bReserved1       : Pos 0, 16 Bits
[SNIPPED FLAGS]
   +0x034 bWS_POPUP        : Pos 31, 1 Bit
   +0x038 hModule          : Ptr64 Void
   +0x040 hMod16           : Uint2B
   +0x042 fnid             : Uint2B
   +0x048 spwndNext        : Ptr64 tagWND
   +0x050 spwndPrev        : Ptr64 tagWND
   +0x058 spwndParent      : Ptr64 tagWND
   +0x060 spwndChild       : Ptr64 tagWND
   +0x068 spwndOwner       : Ptr64 tagWND
   +0x070 rcWindow         : tagRECT
   +0x080 rcClient         : tagRECT
   +0x090 lpfnWndProc      : Ptr64     int64
   +0x098 pcls             : Ptr64 tagCLS
   +0x0a0 hrgnUpdate       : Ptr64 HRGN__
   +0x0a8 ppropList        : Ptr64 tagPROPLIST
   +0x0b0 pSBInfo          : Ptr64 tagSBINFO
   +0x0b8 spmenuSys        : Ptr64 tagMENU
   +0x0c0 spmenu           : Ptr64 tagMENU
   +0x0c8 hrgnClip         : Ptr64 HRGN__
   +0x0d0 hrgnNewFrame     : Ptr64 HRGN__
   +0x0d8 strName          : _LARGE_UNICODE_STRING
   +0x0e8 cbwndExtra       : Int4B
   +0x0f0 spwndLastActive  : Ptr64 tagWND
   +0x0f8 hImc             : Ptr64 HIMC__
   +0x100 dwUserData       : Uint8B
   +0x108 pActCtx          : Ptr64 _ACTIVATION_CONTEXT
   +0x110 pTransform       : Ptr64 _D3DMATRIX
   +0x118 spwndClipboardListenerNext : Ptr64 tagWND
   +0x120 ExStyle2         : Uint4B
   +0x120 bClipboardListener : Pos 0, 1 Bit
[SNIPPED FLAGS]
   +0x120 bChildNoActivate : Pos 11, 1 Bit
```

In the 64-bit structure above we can see that the `_LARGE_UNICODE_STRING` structure we want to overwrite starts at offset 0xd8. You'll also notice a significant number of members earlier in the structure. Originally I had hoped to just trample this carefree, but there are numerous pointers in `_THRDESKHEAD` that we need to stay sane, and unfortunately we can't actually control what we write there because of the constraints we already discussed.

The `_THRDESKHEAD` structure looks like:

```
kd> dt !_THRDESKHEAD
win32k!_THRDESKHEAD
   +0x000 h                : Ptr64 Void
   +0x008 cLockObj         : Uint4B
   +0x010 pti              : Ptr64 tagTHREADINFO
   +0x018 rpdesk           : Ptr64 tagDESKTOP
   +0x020 pSelf            : Ptr64 UChar
```

Not only does clobbering `_THRDESKHEAD` cause us problems, but let's revisit our alignment constraints. Our new `tagPROP` entry at whatever offset we are writing it will always end up writing directly over top of the exact start of the `_LARGE_UNICODE_STRING` structure:

```
win32k!_LARGE_UNICODE_STRING
   +0x000 Length           <-- hData (fully controlled) would
overwrite this
   +0x004 MaximumLength    <-- and this
   +0x004 bAnsi            <-- and this
   +0x008 Buffer           <-- atomKey and fs (only partially
controlled) would overwrite this
```

It's specifically the `Buffer` pointer we want to overwrite in order to access arbitrary memory however, so even if we could safely clobber the rest of the structure, we wouldn't control the one pointer we need to control.

The answer to our inability to corrupt arbitrary data is to turn the `tagPROPLIST` corruption into an entirely different corruption mechanism.

On versions of Windows after XP, the userland backend allocator (and so kernel desktop heap) chunk headers (aka `_HEAP_ENTRY` structures) are stored in-band and are located right before the actual contents of the chunk. The Desktop heap itself is managed by a `_HEAP` structure, which tracks the various free and in use chunks.

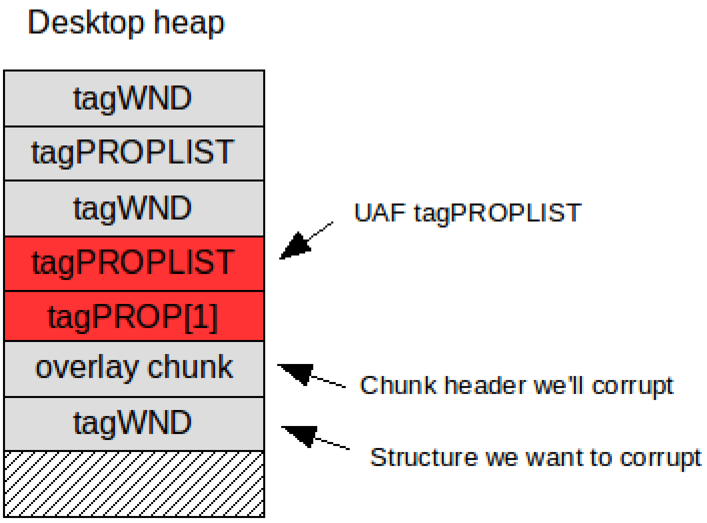
A `_HEAP_ENTRY` is defined as follows:

```
kd> dt !_HEAP_ENTRY
ntdll!_HEAP_ENTRY
   +0x000 PreviousBlockPrivateData : Ptr64 Void
   +0x008 Size                   : Uint2B
   +0x00a Flags                  : UChar
   +0x00b SmallTagIndex          : UChar
   +0x00c PreviousSize           : Uint2B
   +0x00e SegmentOffset          : UChar
   +0x00f UnusedBytes            : UChar
```

The chunk header is 0x10 bytes total. The first eight bytes, called `PreviousBlockPrivateData`, are used to hold actual chunk data from a previous chunk if the requested size spilled over the normal 0x10 chunk alignment by <= 8 bytes. This is explained briefly in a nice Leviathan blog entry, as well as other earlier userland heap articles. The `Size` and `PreviousSize` members represent the chunk size of the current and previous chunks, divided by 0x10. The `Flags` member is used to indicate if a given chunk is free, etc. If `_HEAP_ENTRY` security is enabled in the corresponding `_HEAP` structure that manages the heap, then the `SmallTagIndex` entry will hold a XORed checksum value of some of the expected values in the chunk.

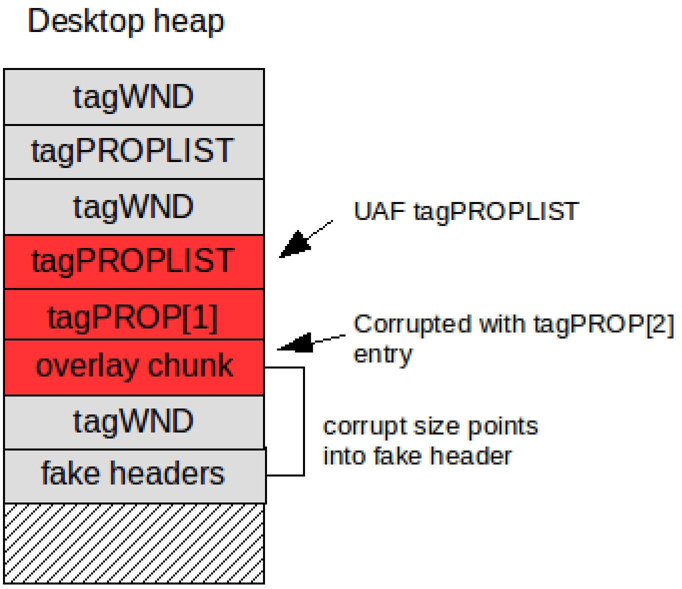
Although alignment hasn't been favorable to us so far, it is actually in this situation. If you recall the `tagPROPLIST` is always at least 0x18, and then an extra 0x10 bytes for every new `tagPROP` entry added. For a two-entry property list of size 0x28 this means it will actually be placed into a chunk of 0x20 bytes and those `PreviousBlockPrivateData` spillover bytes are used from the adjacent chunk. And this means that when we add a third entry and corrupt whatever is immediately adjacent, the eight bytes of `hData` bytes we control will fit exactly over the top of the more interesting parts of the `_HEAP_ENTRY` structure.

What we want to do is abuse this so that we can somehow write arbitrary data over the top of the `Buffer` address. First we modify our heap layout so that adjacent to our corrupted `tagPROPLIST` chunk we have a small chunk containing a text string associated with some window we control during setup. I refer to this as an overlay chunk. Adjacent to this overlay chunk we place the `tagWND` structure we actually want to corrupt. This is illustrated below. Note that I've begun to omit the earlier sprayed chunks to save size, so these should now be taken as implied.

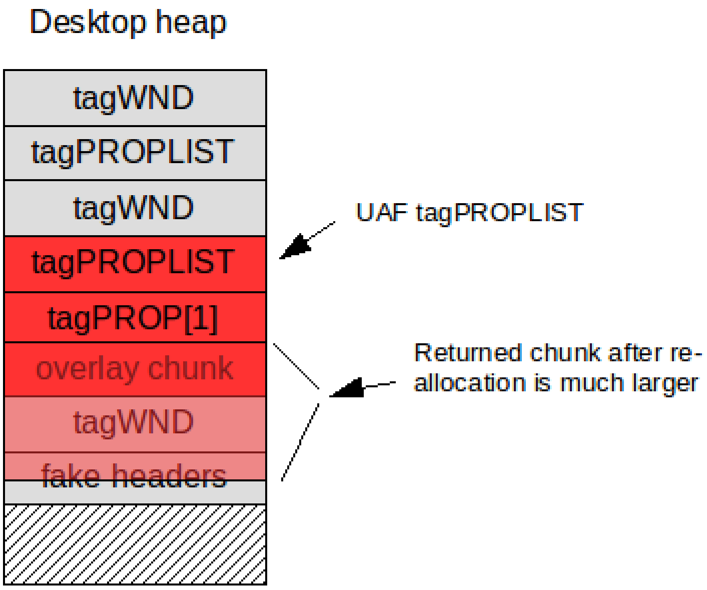


Next we insert a third `tagPROP` entry into our corrupted `tagPROPLIST` list, which overwrites the last eight bytes of the `_HEAP_ENTRY` and the first eight bytes of whatever was in the overlay chunk. When we modify the `_HEAP_ENTRY` we specifically modify the `Size` parameter of the overlay chunk to be significantly larger than the actual chunk size, making it large enough that it will include (or overlay) the adjacent `tagWND` structure.

The goal is now to free the overlay chunk we just corrupted so that the heap algorithm places it on a free list associated with a size that is larger than the chunk actually is, and then reallocates it for use with new window text, which we will fully control. However, this causes a small problem we first need to deal with. When the chunk is being freed, the heap algorithm will try to walk ahead on the heap to the next adjacent chunk, which it determines using the corrupted `Size` member. It will try to determine if this adjacent chunk is free and if so will attempt to coalesce it. We want to make sure we control whatever it tries to reference, and that a flag indicating that the chunk is busy is set. We do this by again slightly modifying our heap layout. This time we place a buffer of fake heap headers all with the busy flag set and also with `PreviousSize` values that correspond to the corrupted `Size`, which we can do simply by using another window text allocation associated with yet another window This new layout is illustrated as follows:

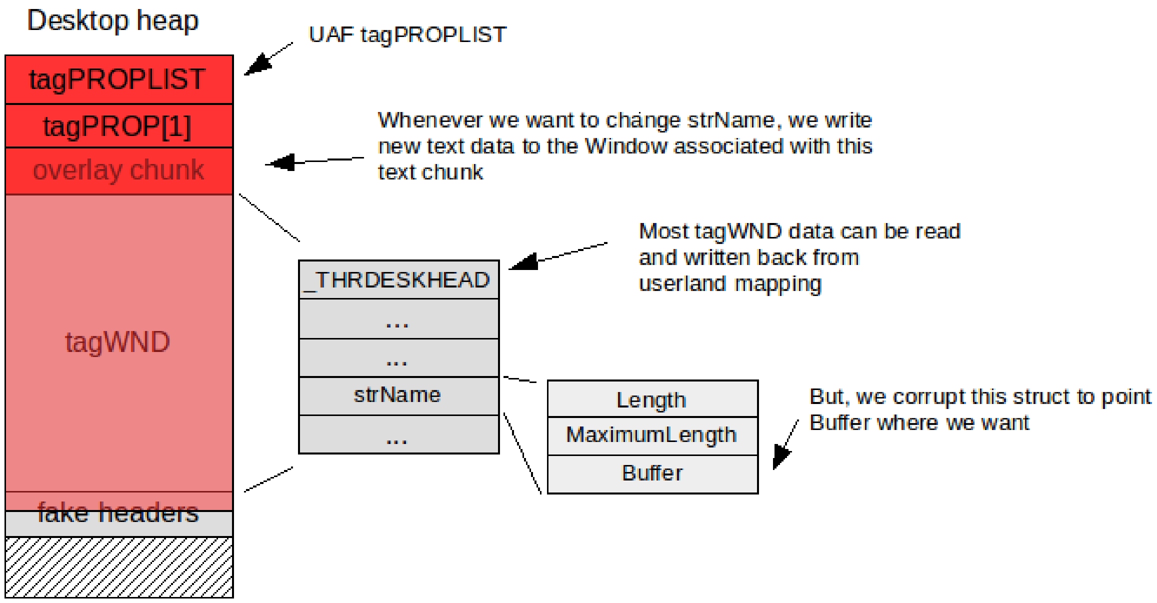


Now finally we can free the corrupted overlay chunk by updating the associated window with a larger string than was originally allocated (0x10 bytes in the diagrams). This will first free the corrupted chunk, placing it on the free list. However, the size was corrupted, and now the freed chunk is advertised as much larger than it actually is. So we can actually just force this newly freed chunk to be reused to service our new chunk allocation of larger size. This results in our string data being written to the chunk, which we can then use to corrupt all of the adjacent `tagWND` structure with arbitrary data. This is illustrated in the diagrams below:



So that facilitates what I describe as stage three corruption. We can technically now overwrite the `strName.Buffer` pointer with any data we want. However, we still have the issue of corrupting everything else in `tagWND` beforehand. But it turns out this isn't a problem, because the desktop heap is mapped to userland! So before we corrupt everything, we just read all of the contents out of the target `tagWND` structure, modify the `strName` structure contents to anything we want, and then send all that data through as our text update!

Not only does this give us an arbitrary read or write primitive via the `strName` structure, but updating `strName` is repeatable because of the way the logic of window text updating works. As long as the string you're writing into the buffer is <= the `MaximumLength` member, it will continue to reuse the same chunk. So every time we want to change the address of `strName` to read or write somewhere new, we re-update the overlay chunk with a new string and resupply our arbitrary data. The repeatability is illustrated in the diagram below. Note that I once again zoom in the graphic for greater granularity on what is being corrupted each time:


Desktop heap
tagPROPLIST
tagPROP[1]
overlay chunk
tagWND
fake headers
UAF tagPROPLIST
Whenever we want to change strName, we write new text data to the Window associated with this text chunk
_THRDESKHEAD
...
...
strName
...
Most tagWND data can be read and written back from userland mapping
Length
MaximumLength
Buffer
But, we corrupt this struct to point Buffer where we want

This means we only end up corrupting two additional things (aside from the original `tagPROPLIST` entry):

1. The overlay chunk heap header. We can actually read this before we corrupt it, so we know how to fix it up after the fact. Funnily enough, we can even fix up the chunk by rewriting the 3rd `tagPROPLIST` entry, as long as we send through the `atomKey` we used in order to corrupt it in the first place!
2. The `strName` structure, which we can easily fix by a subsequent write of window text data. We can just set this all back to NULL when we're done.

So now if we want to read some amount of bytes from anywhere in memory, we query the window text, using `InternalGetWindowText()`, that is associated with the target window that has the corrupted `strName` entry. We can read up to the number of bytes we placed into the `Length` member. Similarly, if we want to write to an arbitrary location in memory, we update that corrupted window's text, using `NtUserDefSetText` with data equal to or less than the fake size we placed into the `MaximumLength` member, and it will reuse the existing buffer, which just points wherever we want.

Windows 8 and 8.1 heap encoding

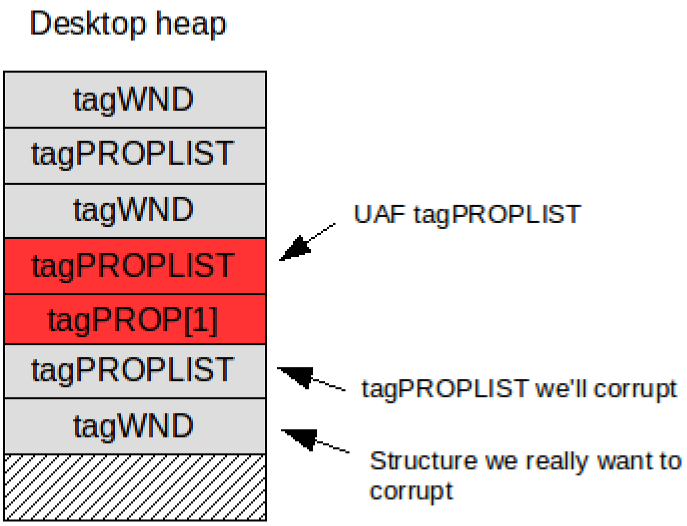
Although the backend allocator in userland started using heap encoding as of Windows Vista, the desktop heap never bothered enabling it until Windows 8 and later. So this causes a hurdle when we're doing the overlay chunk overwrite that was described above. However, it turns out the `_HEAP` structure used to define the heap also holds the actual cookie used to encode all heap headers, so we can just read this out of the userland mapping of the desktop heap and then ensure that the overlay chunks header is properly encoded, by mimicking the logic reversed from the allocator, and the allocator won't complain.

32-bit

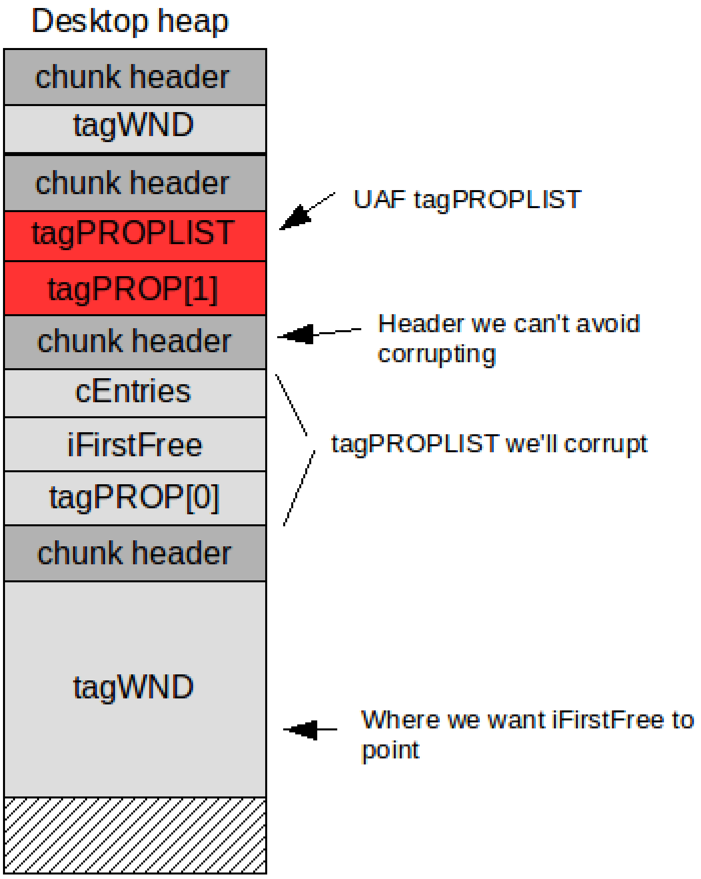
The first and most important thing to note about 32-bit is that the `tagPROP` structure is now a total of 0x8 bytes (instead of 0xc bytes on 64-bit), and the `hData` member we control is now only 0x4 bytes (rather than 0x8 bytes on 64-bit). Also there is no longer any additional padding (instead of the 0x4 bytes of padding on 64-bit), as the whole structure fits neatly within 0x8 bytes. But this means that we can't completely corrupt an adjacent chunk header as we would only partially control the data. On some versions of Windows this would be okay, as we can control the most important fields, but on Windows 8 and 8.1 where the header is encoded, we will end up overwriting a part of the header with the `fs` member in an unsafe way. On 32-bit the `_HEAP_ENTRY` header looks similar, but is missing the `PreviousBlockPrivateData` entry.

We still can't corrupt every part of `tagWND` because we inevitably break pointers. But I still didn't find any new interesting objects to target and because `_LARGE_UNICODE_STRING` worked so well on 64-bit I was kind of set on using it for 32-bit.

My idea was that if we could corrupt the `iFirstFree` member (recall, the index of the first free entry in the property list) of a `tagPROPLIST` structure in such a way that we could increase the index, we could point it to some adjacent location further on the heap. We could, for instance, try to point it over the top of the `strName` member of the adjacent `tagWND` struct. The following diagram illustrates this general idea:



To make things clear now that we're dealing with two `tagPROPLIST` structures, I'll call the UAF property list 'list A' and the other list 'list B'. We need to know exactly what part of our `tagPROP` inserted into 'list A' will overwrite the `iFirstFree` entry of 'list B'. We also have to keep in mind that we're only writing eight bytes at a time, so we're going to have to at least insert one additional `tagPROP` into 'list A', as the first will corrupt the adjacent heap header, and only the second will hit the 'list B' `tagPROPLIST` values. Depending on the OS and chunk sizes in play, this may vary, and I had to accommodate various layouts in my exploit. For now let's assume that we need to insert only two to corrupt 'list B' though. What exactly we'll be corrupting is shown in the following diagram. Note that the first `tagPROPLIST` in the diagram is not broken out into its individual members, so `tagPROP[0]` is implied. However, in the second `tagPROPLIST` I have broken out the internals to show what we're corrupting, which is why the `tagPROP[0]` entry is shown:



First we note that if we're writing eight bytes for each `tagPROP`, then that means we'll only partially control what is written over `iFirstFree` (because it will come from `atomKey` and `fs` members), which is the value we're most concerned about. Because we fully control the two least significant bytes of the value with our `atomKey` it actually works out okay, as the value will be small enough that `fs` will be 0. So we will use our `hData` value to overwrite `cEntries` with some sane value, and use `atomKey` to point `iFirstFree` where we want in the target `tagWND`. We want to overwrite the `strName.Buffer` pointer in `tagWND` specifically. If we couldn't overwrite the `Length` and `MaximumLength` values directly, that would be okay, because we could pre-allocate a string for the target window to ensure the lengths are already set to some large value.

Let's look at the `tagWND` struct on 32-bit to see what we've got. Note that this time I'm using the `-b` switch so we can easily compute the offset of the `Buffer` embedded in the `strName` struct.

```
kd> dt -b !tagWND
win32k!tagWND
   +0x000 head             : _THRDESKHEAD
      +0x000 h                : Ptr32
      +0x004 cLockObj         : Uint4B
      +0x008 pti              : Ptr32
      +0x00c rpdesk           : Ptr32
      +0x010 pSelf            : Ptr32
   +0x014 state            : Uint4B
   +0x014 bHasMeun         : Pos 0, 1 Bit
[SNIPPED FLAGS]
   +0x014 bDestroyed       : Pos 31, 1 Bit
   +0x018 state2           : Uint4B
[SNIPPED FLAGS]
   +0x018 bWMCreateMsgProcessed : Pos 31, 1 Bit
   +0x01c ExStyle          : Uint4B
   +0x01c bWS_EX_DLGMODALFRAME : Pos 0, 1 Bit
[SNIPPED FLAGS]
   +0x01c bUIStateFocusRectHidden : Pos 31, 1 Bit
   +0x020 style            : Uint4B
   +0x020 bReserved1       : Pos 0, 16 Bits
[SNIPPED FLAGS]
   +0x020 bWS_POPUP        : Pos 31, 1 Bit
   +0x024 hModule          : Ptr32
   +0x028 hMod16           : Uint2B
   +0x02a fnid             : Uint2B
   +0x02c spwndNext        : Ptr32
   +0x030 spwndPrev        : Ptr32
   +0x034 spwndParent      : Ptr32
   +0x038 spwndChild       : Ptr32
   +0x03c spwndOwner       : Ptr32
   +0x040 rcWindow         : tagRECT
      +0x000 left             : Int4B
      +0x004 top              : Int4B
      +0x008 right            : Int4B
      +0x00c bottom           : Int4B
   +0x050 rcClient         : tagRECT
      +0x000 left             : Int4B
      +0x004 top              : Int4B
      +0x008 right            : Int4B
      +0x00c bottom           : Int4B
   +0x060 lpfnWndProc      : Ptr32
   +0x064 pcls             : Ptr32
   +0x068 hrgnUpdate       : Ptr32
   +0x06c ppropList        : Ptr32
   +0x070 pSBInfo          : Ptr32
   +0x074 spmenuSys        : Ptr32
   +0x078 spmenu           : Ptr32
   +0x07c hrgnClip         : Ptr32
   +0x080 hrgnNewFrame     : Ptr32
   +0x084 strName          : _LARGE_UNICODE_STRING
      +0x000 Length           : Uint4B
      +0x004 MaximumLength    : Pos 0, 31 Bits
      +0x004 bAnsi            : Pos 31, 1 Bit
      +0x008 Buffer           : Ptr32
   +0x090 cbwndExtra       : Int4B
   +0x094 spwndLastActive  : Ptr32
   +0x098 hImc             : Ptr32
   +0x09c dwUserData       : Uint4B
   +0x0a0 pActCtx          : Ptr32
   +0x0a4 pTransform       : Ptr32
   +0x0a8 spwndClipboardListenerNext : Ptr32
   +0x0ac ExStyle2         : Uint4B
   +0x0ac bClipboardListener : Pos 0, 1 Bit
[SNIPPED FLAGS]
   +0x0ac bChildNoActivate : Pos 11, 1 Bit
```

We see that `strName` is at offset 0x84 and `Buffer` is at offset 0x8c specifically. Remember we're indexing from a 0x8 byte aligned array of `tagPROP` entries and we can only write 0x8 bytes. So we can easily work out that if we chose to make the `iFirstFree` index to offset 0x88 in the `tagWND` we would point at `MaximumLength` (overwritten by `hData`) and our write wouldn't work because we'd only control two bytes of `Buffer`, whereas we want this to be our arbitrary read/write primitive, so this isn't acceptable. If we try to write to the next index and point to 0x90 then we'll be overwriting `cbwndExtra` (with `hData`), which isn't what we're after.

We need to think back to earlier and what we control for the purposes of doing heap feng shui, and then look at these in `tagWND` to see if anything is at interesting offsets we might control. At offset 0x70 in the `tagWND` struct we see the `pSBInfo`. This is divisible by 0x8 so we know that we would actually be able to overwrite this pointer with the `hData` portion of our fake `tagPROP`.

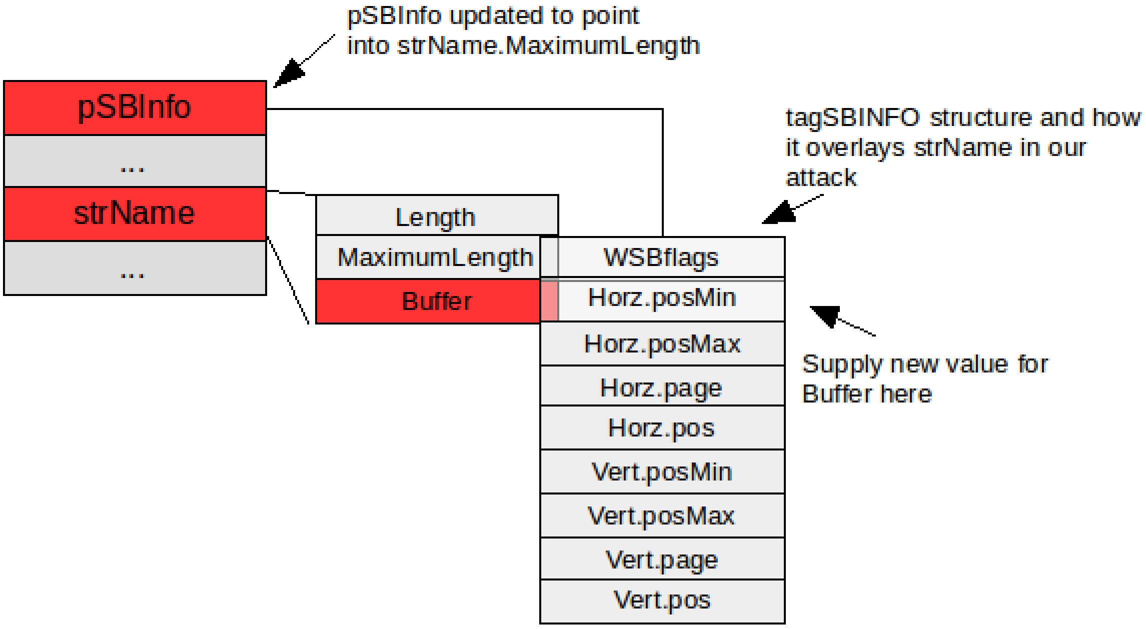
What if we overwrote `pSBInfo` to point directly at `strName` in the same `tagWND` struct? Maybe we could use the scrollbar API to corrupt `strName` to get our primitive.

The `pSBInfo` member points to a `tagSBINFO` struct, which you might recall is the structure type used way back during the very first use-after-free.

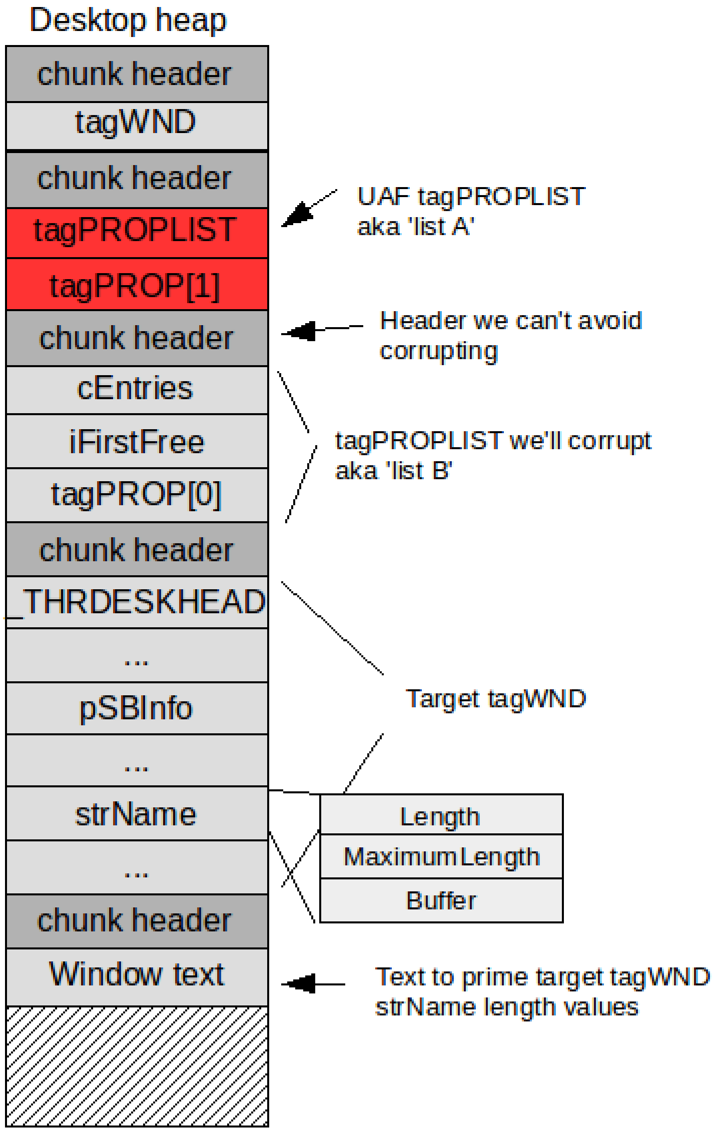
```
kd> dt -b !tagSBINFO
win32k!tagSBINFO
   +0x000 WSBflags         : Int4B
   +0x004 Horz             : tagSBDATA
      +0x000 posMin           : Int4B
      +0x004 posMax           : Int4B
      +0x008 page             : Int4B
      +0x00c pos              : Int4B
   +0x014 Vert             : tagSBDATA
      +0x000 posMin           : Int4B
      +0x004 posMax           : Int4B
      +0x008 page             : Int4B
      +0x00c pos              : Int4B
```

If we remember, the `WSBflags` don't give us a lot of control, but we know at least that if we enable either scrollbar ones will be set, and if we disable an active scrollbar zeroes will be set. The flags member isn't really ideal for setting arbitrary values, but by reversing some related functionality we can see that if we're not changing that state of the scrollbar, those flags won't be changed. The values inside the `tagSBDATA` structure seem more interesting though. If we look at the `SetScrollInfo()` documentation we can get a good understanding of what all of these values represent. It looks like we can set these parameters in a `SCROLLINFO` struct that we pass to `SetScrollInfo()`. As long as the adjacent window is a scrollbar control we are corrupting, it will operate on the `pSBInfo` pointer directly (otherwise it would send a special window message to the associated scrollbar control window). It appears we can control the `posMin` and `posMax` values without any restrictions. The `Page` and `pos` members are a little more finicky as they are expected to be within established range limits, so let’s try to avoid them for now. We will pass the `SIF_RANGE` flag in the `SCROLLINFO`struct to indicate where we want to write the min and max values specifically.

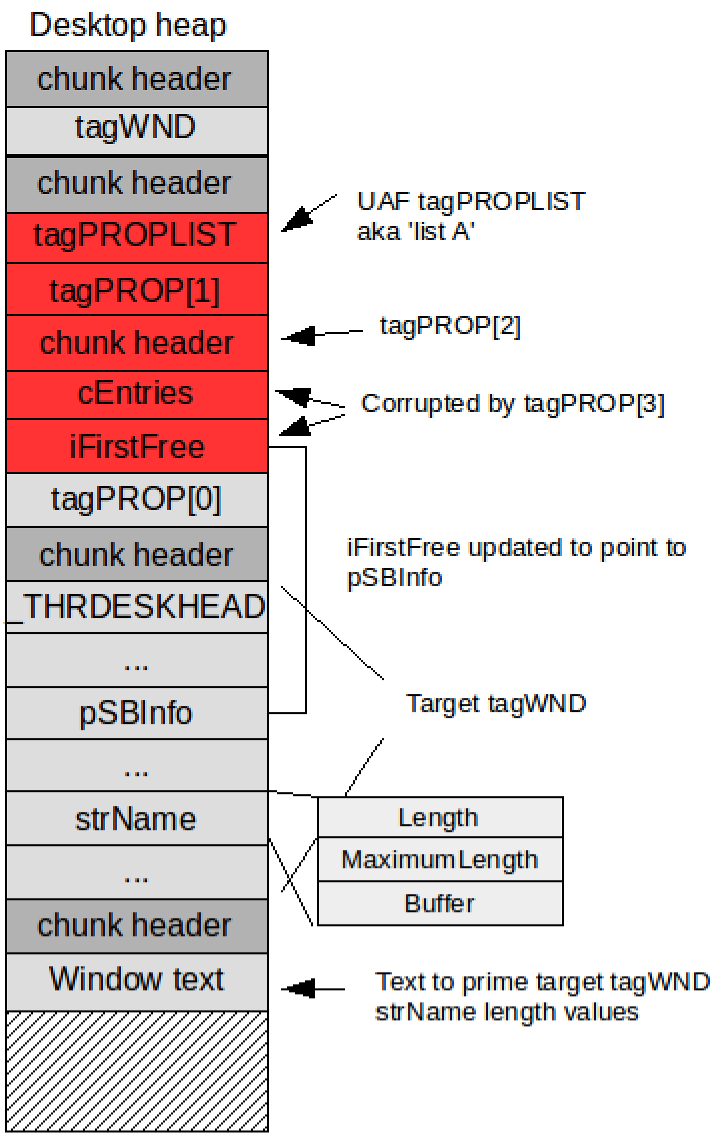
We want to overwrite `Buffer` with arbitrary data, which means we want `posMin` to overlap it, so we can overwrite `pSBInfo` to point to `strName.MaximumLength`. As long as we don't enable or disable the scrollbar, the `WSBflags` member won't be written to, which keeps `strName.MaximumLength` intact. This means that whatever we placed in `posMin`, via `nMin` from `SCROLLINFO`, will be written to `Buffer,` and `posMax` will be written over whatever is next, which is `cbwndExtra`. This isn't a big deal; as with 64-bit, we can pre-read the value and fix it up later. The general concept of the overlap is illustrated below:


pSBInfo updated to point into strName.MaximumLength
pSBInfo
...
strName
...
Length
MaximumLength
Buffer
WSBflags
Horz.posMin
Horz.posMax
Horz.page
Horz.pos
Vert.posMin
Vert.posMax
Vert.page
Vert.pos
tagSBINFO structure and how it overlays strName in our attack
Supply new value for Buffer here

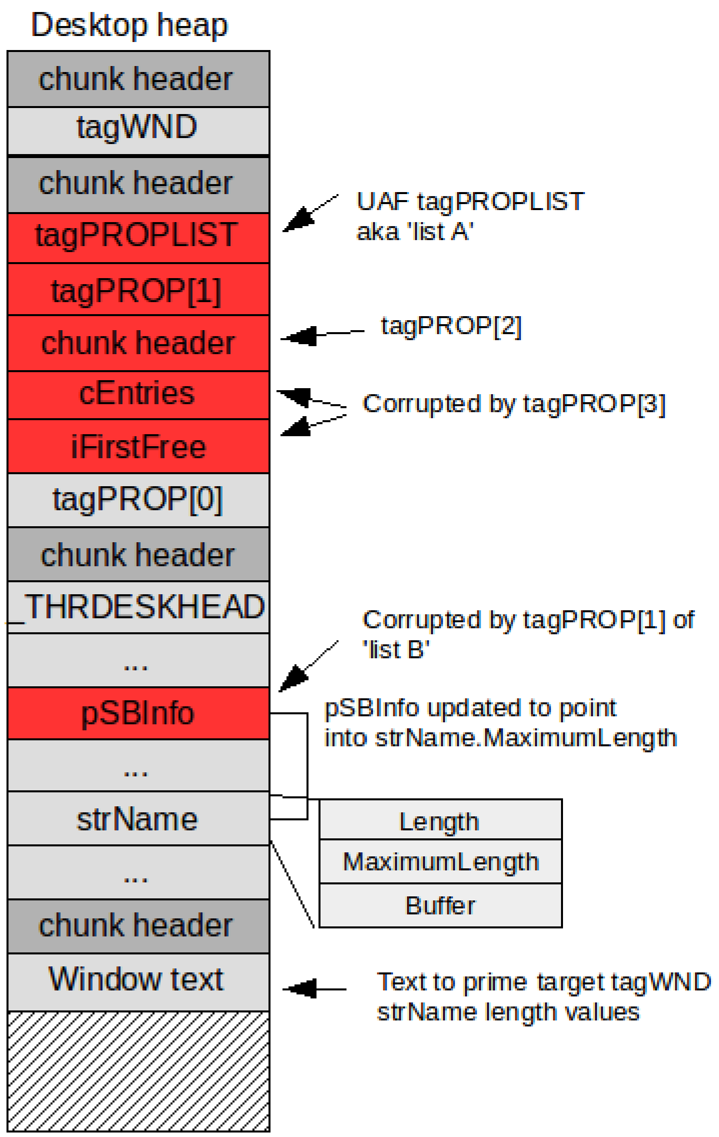
So let's walk through the 32-bit attack with diagrams. First, let's take a step back and look at a diagram showing the relevant chunks and heap layout before we corrupt anything beyond the initial use-after-free. I've now included even more granular info so it's clear exactly what we're doing.



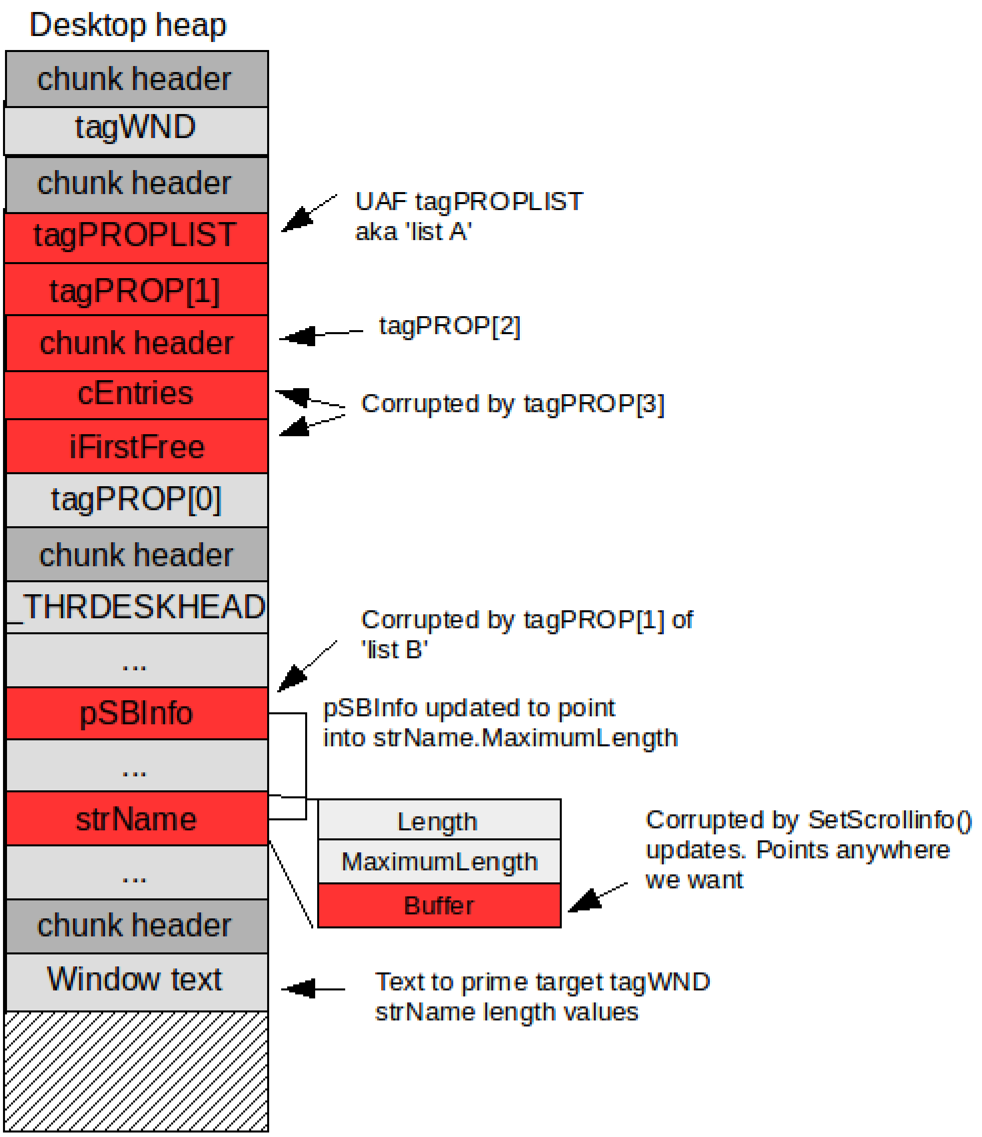
Next we insert two additional entries into 'list A', which will corrupt data adjacent to 'list A' thanks to the use-after-free corruption, and allow us to point 'list B's `iFirstFree` entry to `pSBInfo`. Note that we'll also corrupt the value adjacent to `pSBInfo`, but again we can pre-read it and fix it up later.



Then we insert a new `tagPROP` into 'list B' with an identifier that doesn't already exist in the list resulting in the new entry being inserted at the next free index, which corrupts `pSBInfo` to point at `strName.MaximumLength` in the same `tagWND` struct.



Finally we update the scrollbar structure to corrupt the `Buffer` member:



Remember that unlike the 64-bit case we're not actually corrupting the length values in `strName`, so we need them to be sane already. We can do this by pre-allocating a text buffer with a decently large length for the window, so that the values are already useful. Then whenever we want to read or write a new value in kernel memory at a different address, we simply call `SetScrollInfo()` on the target window and update the location of `Buffer,` and then use the window text API on the target window.

That's it! We now have our repeatable arbitrary read/write primitive on 32-bit!

Getting code exec

Everything from now on simply assumes we have an arbitrary read/write primitive. So if I say leak/read X or overwrite Y, it is just being done with the primitives we have built in the earlier corruption stage. Mostly this is the same on both architectures. All we really have left to do is overwrite a function pointer and redirect it to a payload somewhere. The most common way to do this (popularised by reversemode) is to overwrite the second entry in `nt!HalDispatchTable`, which normally holds the `HalQuerySystemInformation()` function. Then you can force the pointer to be called by calling `NtQueryInternalProfile()` from userland.

We need to leak the base address of the kernel in order to find `nt!HalDispatchTable`. To do this I use the standard `NtQuerySystemInformation()` to fetch module information from which you can pull a base address.

```
        // 11 corresponds to SystemModuleInformation class, which is
undocumented...
        rc = NtQuerySystemInformation((SYSTEM_INFORMATION_CLASS)11,
pModuleInfo, 0x100000, NULL);
```

Then I open a local copy of `ntoskrnl.exe` and look up the `nt!HalDispatchTable` offset, which I can then easily apply to the leaked base address. I then use the read primitive to read the original `HaliQuerySystemInformation()`address (which is not exported) so that it can be fixed up later, and then use the write primitive to corrupt the function pointer with the address of shellcode (in userspace or kernelspace; more on this later). This is the same on both 64-bit and 32-bit aside from the size of the reads and writes.

Bypassing SMEP

SMEP support was introduced on Windows 8 and 8.1, and technically some security products enforce it on Windows 7, so we can assume it's there as well. It prevents us from executing code in userland while operating with kernel privileges, which makes overwriting the `nt!HalDispatchTable` entry to point directly into userland much less useful. So instead we want to point it somewhere in kernelspace that we control, that we can use to disable SMEP by writing to `cr4` and only then jump into userland. MWR documented an interesting trick on 64-bit that involves self-mapping page table entries that let you work out a valid kernel address for an arbitrary virtual address. As long as you have a write primitive you can thus write directly into the page table entry and modify the bits. I was able to adapt this trick to 32-bit fairly easily, although the indices differ between PAE and non-PAE systems.

The most obvious way to do this would be to map an address in userland and then use the write primitive to mark the page table entries as being supervisor rather than user. This is what I tried first; it worked up until Windows 8 and then I ran into an interesting problem. The desktop window manager (`dwm.exe`) on Windows 8 and later regularly scans through windows on desktops and queries their names for some reason (which I didn't investigate). It does this without actually sending them a window message that you can ignore, but instead is able to find

the window handle and calls `GetInternalWindowText()`. So the problem is that we are using the `strName` member of the window structure to point at the page table entry of our memory mapping containing our shellcode, which is private to our processes page table.
When `dwm.exe` asks the kernel to give it the name, the wrong page table entries will be present and so the kernel will see that the `strName.Buffer` address is not NULL and then dereference the address, which will be invalid. This will BSOD the machine.

I worked around this by accepting that `dwm.exe` might query us, and so opting to use a kernel address to hold the payload instead. This way the associated page table entry with that address will always be valid no matter what process is currently loaded. I chose to place it onto the desktop heap since I was already able to compute the kernel address of it using the previously-mentioned approach. We can still use the self-mapping page table entry trick, but in this case the page table is already marked as supervisor, it just won't be marked as executable. So instead we just set the executable bit.

So the steps are quite easy:

1. Create a window text buffer containing our stage one payload and compute the kernel address.
2. Use the self-mapping trick to compute the page table entry for the kernel address from the previous step.
3. Use our write primitive to set the executable bit of the page table entry.
4. Overwrite `nt!HalDispatchTable` to the stage one kernel address.
5. Call `NtQueryInternalProfile()` to jump to the payload.
6. Disable SMEP in cr4 and jump to the stage two userland payload.
7. Execute the stage two userland payload to elevate privileges and return.
8. Restore SMEP in cr4 to prevent patchguard from complaining and return cleanly.

Bypassing Low Integrity Sandbox

On Windows 8.1 we might have an additional problem, which is that the `NtQuerySystemInformation()` function now has checks for low integrity SIDs, which means only medium integrity and above can leak the address of kernel bases. This is actually quite easy to get around using the well-known `sidt` trick. We store the address of the IDT into userland (which is unprivileged) and then use the read primitive to read whatever IDT index we want. Most of them point into the kernel, so we can leak the address of an interrupt handler in the kernel, and then do the exact same PE search.

Once we have the base, we can do the same computation of the `nt!HalDispatchTable` address.

Normally the way you could do this type of lookups is to open `ntokrnl.exe` from the filesystem and then resolve the symbol offset locally and add it to the leaked kernel base. However, in a hardened sandbox this is not ideal because there might be filesystem restrictions imposed preventing you from reading `C:\windows\system32\ntoksrnl.exe` for instance. To work around this type of restriction we can use our leak primitive to parse the symbols we need from the kernel PE header in memory.

Conclusion

That’s all. If you’ve made it this far I genuinely appreciate you taking the time to read everything! In the end, using the techniques described in this write up, I was able to develop a reliable exploit targeting both 32-bit and 64-bit on XP, Vista, 7, 8, 8.1, and Server 2012. On Windows 2003 and 2008 by default theming is not used, and it turns out that theming is required for the usermode callback to hook; so I was unable to exploit these systems unless theming was explicitly enabled.

The process of exploitation was quite complicated and there were many hurdles to overcome, but it also prompted a lot of interesting findings, a ton of learning, and was a great way to validate and more thoroughly understand what many other public researchers hint at in their papers.

There is only one mitigation to stop this type of win32k.sys vulnerabilities to my knowledge, which the Google Chrome sandbox uses, which is to effectively disable win32k syscalls at runtime.

As always I appreciate any feedback or corrections. If I described some technique and didn’t adequately provide credit I would also appreciate knowing so I can update the entry. You can contact me via twitter @fidgetingbits or via email: aaron<dot>adams<at>nccgroup<dot>trust.