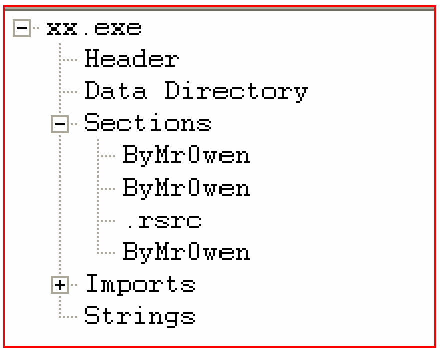
Analysis of ANI "anih" Header Stack Overflow Vulnerability, Microsoft Security Advisory 935423

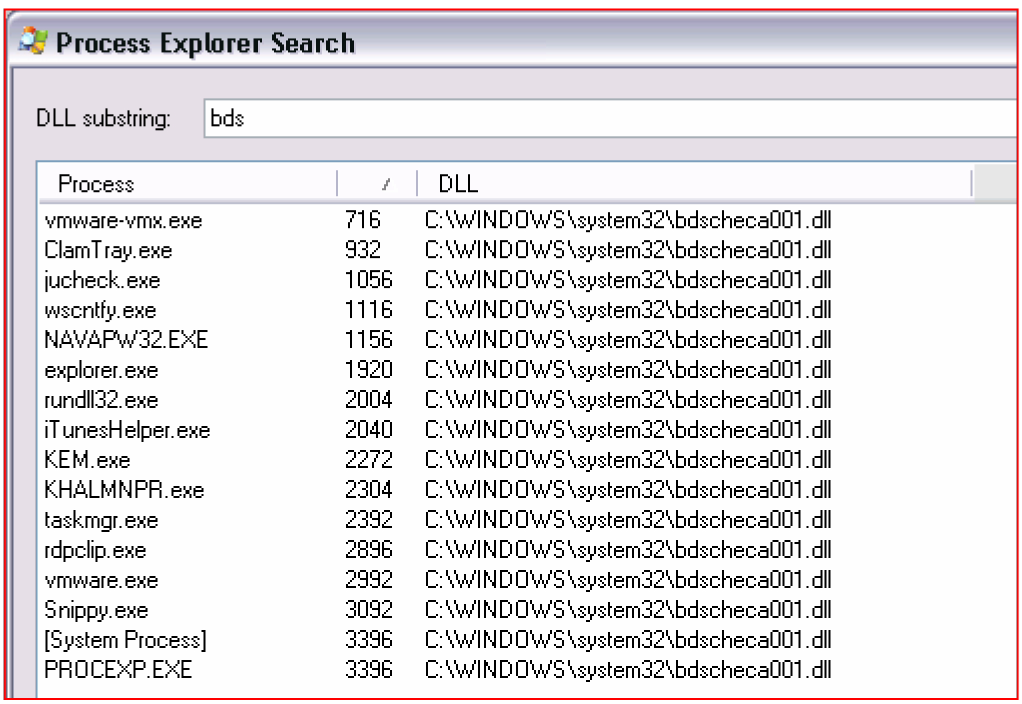
With help from Andre (Dre) and
various other members of mal-aware

This is a really, really rough explanation of the exploit and the actions performed to research the vulnerability.

There are several malicious ANI files in circulation. The one to discuss is mm.jpg from newasp, but others are likely very similar. Shellcode in mm.jpg basically resolves kernel32 functions, downloads, and executes xx.exe (from behavioral analysis). It doesn't do much but delete the system's HOSTS file, write bdscheca001.dll to %SYSTEM%, and registers the DLL as ShellExecuteHooks entry.

This means whenever a process calls ShellExecute() or ShellExecuteEx(), the new DLL will be loaded into that process' address space and its startup routine will be executed. So practically everything is going to call one of these two functions eventually. It will result in all processes being trojanized. Here is a view of the xx.exe sections (who is MrOwen?) and the hooked process list.




Process Explorer Search

DLL substring: bds

Process		DLL
vmware-vmx.exe	716	C:\WINDOWS\system32\bdscheca001.dll
ClamTray.exe	932	C:\WINDOWS\system32\bdscheca001.dll
jucheck.exe	1056	C:\WINDOWS\system32\bdscheca001.dll
wscntfy.exe	1116	C:\WINDOWS\system32\bdscheca001.dll
NAVAPW32.EXE	1156	C:\WINDOWS\system32\bdscheca001.dll
explorer.exe	1920	C:\WINDOWS\system32\bdscheca001.dll
rundll32.exe	2004	C:\WINDOWS\system32\bdscheca001.dll
iTunesHelper.exe	2040	C:\WINDOWS\system32\bdscheca001.dll
KEM.exe	2272	C:\WINDOWS\system32\bdscheca001.dll
KHALMNPR.exe	2304	C:\WINDOWS\system32\bdscheca001.dll
taskmgr.exe	2392	C:\WINDOWS\system32\bdscheca001.dll
rdpclip.exe	2896	C:\WINDOWS\system32\bdscheca001.dll
vmware.exe	2992	C:\WINDOWS\system32\bdscheca001.dll
Snippy.exe	3092	C:\WINDOWS\system32\bdscheca001.dll
[System Process]	3396	C:\WINDOWS\system32\bdscheca001.dll
PROCEXP.EXE	3396	C:\WINDOWS\system32\bdscheca001.dll

Anyway, the point is to find the vulnerability, not analyze the payload, so I have no idea what bdscheca001.dll does. While waiting on a few things, we looked in the mm.jpg at the shellcode. It's pretty obvious where it starts in the file – it starts around 0xA8.

```
00000050  61 6E 69 68  52 00 00 00  30 31 32 33  30 31 32 33  anihR   01230123
00000060  30 31 32 33  30 31 32 33  30 31 32 33  30 31 32 33  0123012301230123
00000070  30 31 32 33  30 31 32 33  00 34 34 34  34 34 34 34  01230123 4444444
00000080  34 34 34 34  34 34 34 34  34 34 34 34  34 34 34 34  4444444444444444
00000090  34 34 34 34  00 00 00 00  00 00 00 00  00 00 00 00  4444
000000A0  00 00 00 00  00 00 00 00  BA 25 8B EC  64 8B 15 30          º%‹ìd‹ 0
000000B0  00 00 00 8D  52 03 80 3A  01 0F 84 C8  00 00 00 C6     R €:  „È   Æ
000000C0  02 01 E8 4B  01 00 00 68  00 03 00 00  6A 00 FF D0    èK   h    j ÿÐ
```

So do a little translation with IDA and now we know that it just resolves Kernel32 exports. Most important, it finds GlobalAlloc() before any others, and allocates 0x300 bytes of memory.

```
seg000:000000A8                 mov     edx, 64EC8B25h
seg000:000000AD                 mov     edx, ds:dword_30
seg000:000000B3                 lea     edx, [edx+3]
seg000:000000B6                 cmp     byte ptr [edx], 1
seg000:000000B9                 jz      loc_187
seg000:000000BF                 mov     byte ptr [edx], 1
seg000:000000C2                 call    GetGlobalAllocAddress
seg000:000000C7                 push    300h
seg000:000000CC                 push    0
seg000:000000CE                 call    eax             ; GlobalAlloc(0,0x300)
seg000:000000D0                 mov     ecx, 300h
seg000:000000D5                 mov     edi, eax
seg000:000000D7                 jmp     short loc_DE
```

So, now we have a good start. Opening IE in a debugger, a breakpoint is set on GlobalAlloc(). Now we can access the HTML file that loads the malicious cursor. There are easier ways, like maybe setting 0xCC as the first byte at 0xA8 offset and catching a break there, but as we will learn soon, this will really break things bad. There are about 30 calls to GlobalAlloc() that we just have to play past until eventually we encounter one where the memory size argument is 0x300, just like in the shellcode. This isn't foolproof (maybe other calls allocate 0x300 bytes, but in our case, the first encounter is the correct one.

If we "execute until return" once inside GlobalAlloc(), our EIP ends up on the heap. The question is – how did we get here…why is EIP on the heap. If we look down on the stack, some return addresses are inside user32.dll.

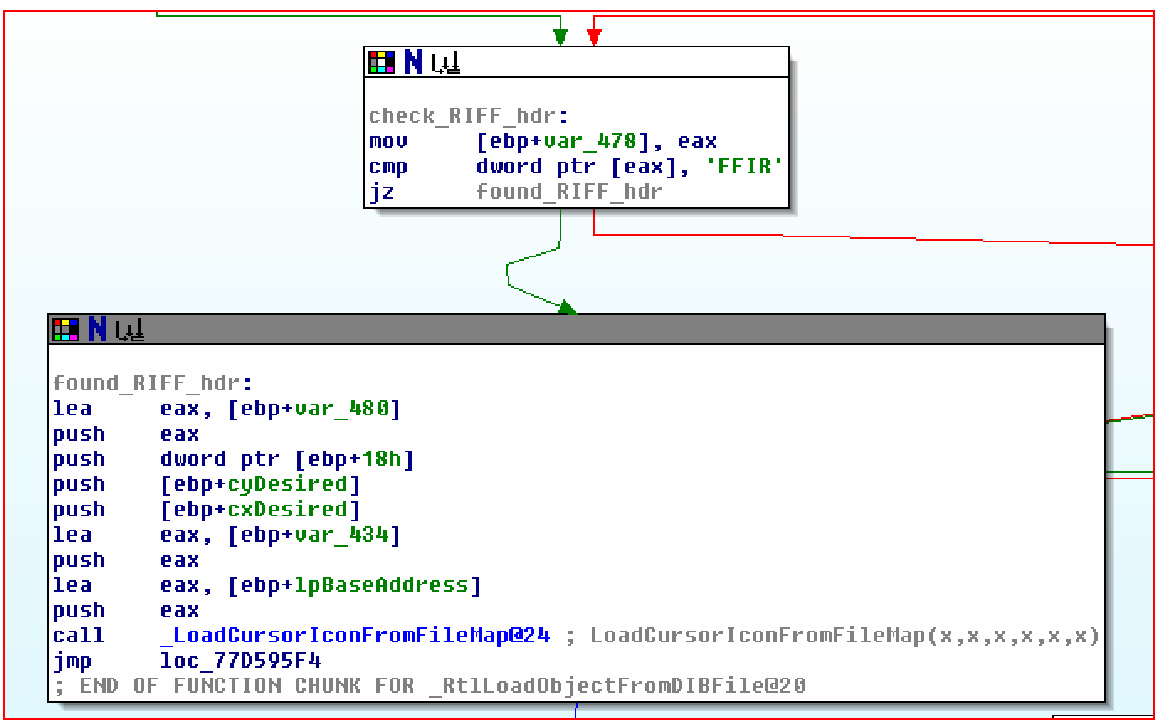
```
0013CF30  02C800D0  ┌CALL to GlobalAlloc from 02C800CE
0013CF34  00000000  │Flags = GMEM_FIXED
0013CF38  00000300  └MemSize = 300 (768.)
0013CF3C  77D825BD  RETURN to USER32.77D825BD
0013CF40  0013D004
0013CF44  00000610
0013CF48  00000000
0013CF4C  00000024
0013CF50  0000FFFF
0013CF54  00000009
0013CF58  00000000
0013CF5C  00000000
```

So let's see what is at this address inside user32.dll:

```
.text:77D825A8                 jz      short loc_77D825C1
.text:77D825AA                 push    eax
.text:77D825AB                 push    [ebp+arg_8]
.text:77D825AE                 push    esi
.text:77D825AF                 call    _MLLine@8       ; MLLine(x,x)
.text:77D825B4                 push    eax
.text:77D825B5                 push    edi
.text:77D825B6                 push    [ebp+arg_4]
.text:77D825B9                 push    esi
.text:77D825BA                 call    dword ptr [ebx+4]
.text:77D825BD                 mov     edi, eax
```

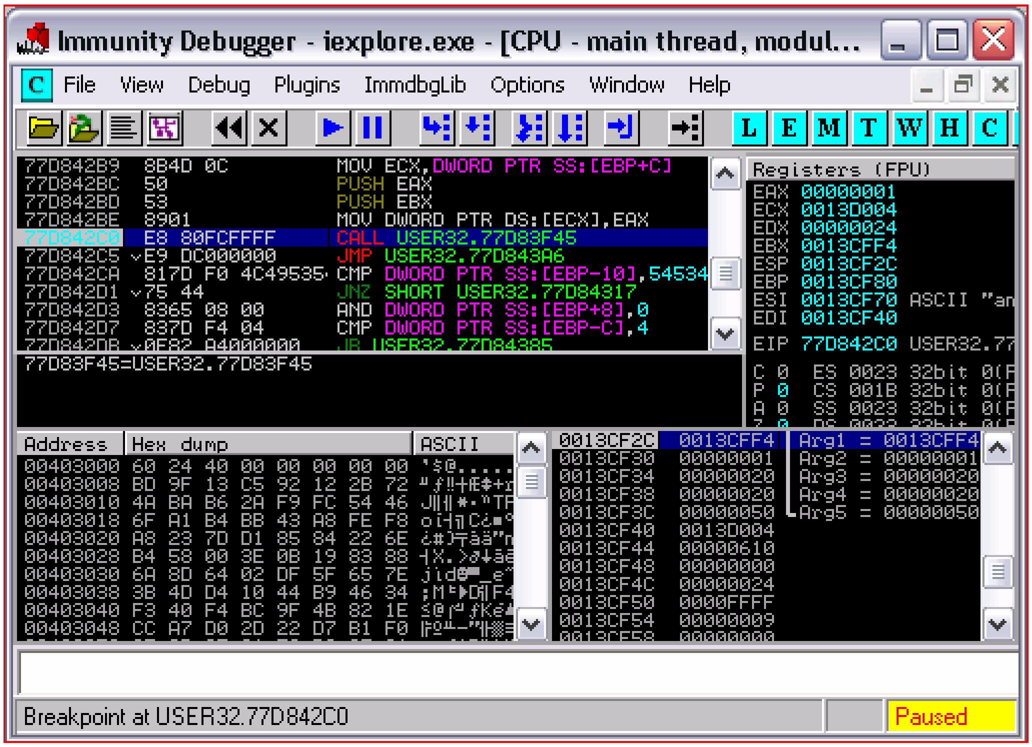
Basically, there is a call [ebx+4] right before the return address we know. So this indicates that EIP was at this call sometime close before it got redirected to the heap. My first assumption is that the vulnerability is inside _MLLine or inside the parent function. But, putting breakpoints on either function fails, and the whole exploit still works. So strangely, this means that somehow EIP ends up at the call [ebx+4], in the middle of the function, without ever starting at the beginning of the parent function.

From here, we back-traced a little to return addresses further down on the stack and found the point where user32.dll opens our mm.jpg and reads it using MapViewOfFile(). It happens inside the RtlLoadObjectFromDIBFile() function. We can confirm, because after mapping the file, it checks the headers of the alleged ANI file. If the header is valid, it goes on to call LoadCursorIconFromFileMap().

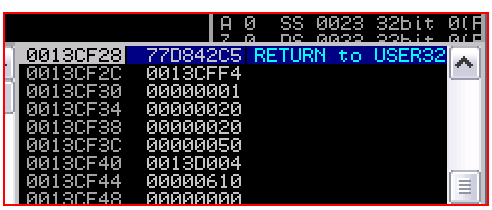


Its important to note that jumping over the LoadCursorIconFromFileMap() function from within RtlLoadObjectFromDIBFile() will execute the exploit. This is good verification though that the vulnerability is getting closer – now we know it exists inside LoadCursorIconFromFileMap() and not some other function that RtlLoadObjectFromDIBFile() calls.

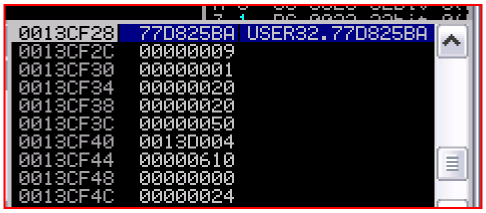
With a little more back-trace, we can narrow it down even further. It turns out that LoadCursorIconFromFileMap() calls a sub function too (several, actually) called LoadAniIcon(). Check out the state of things right before the call to this sub function. Notice how the next instruction after the call is 0x77D842C5. This *should* be the return address when LoadAniIcon() finishes.



Just for a sanity check, step inside the function and see what is pushed on the stack:

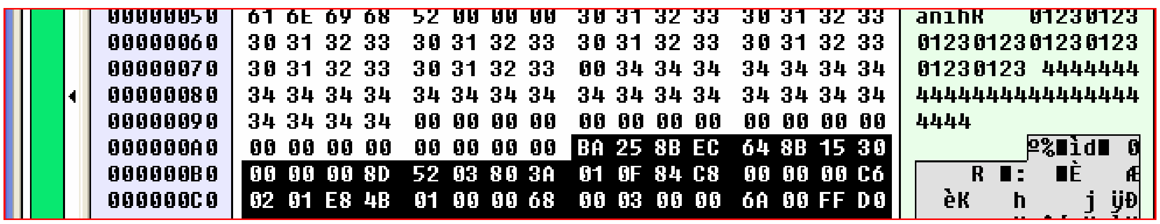


Yep, we should return to 0x77D842C5. Let's fast forward a bit by using "execute until return" and take another look at the stack.



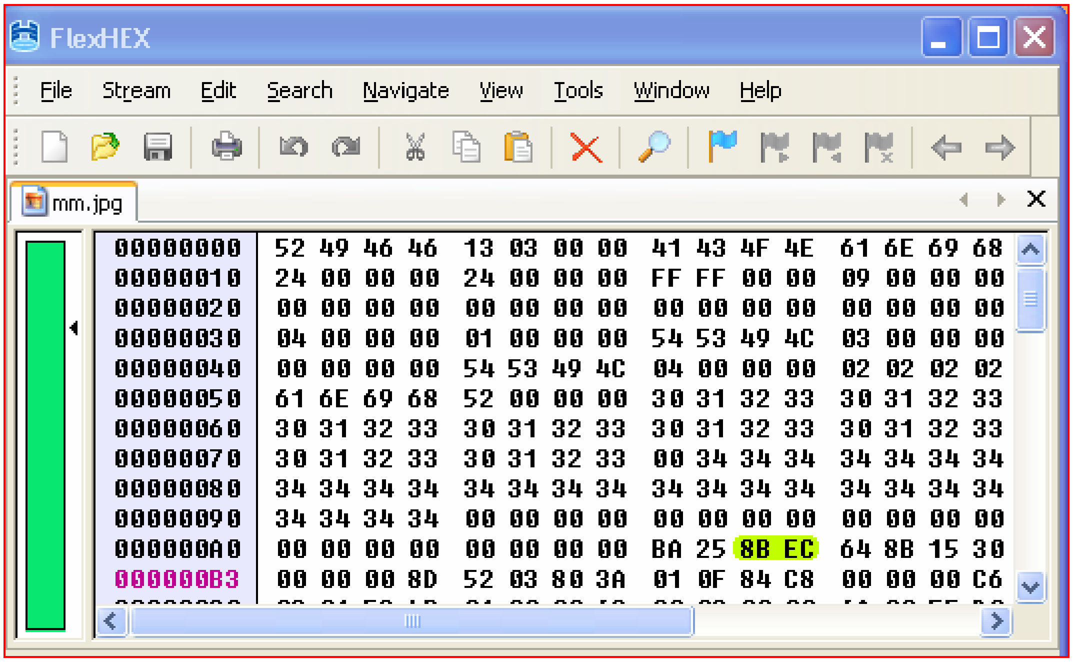
So, the bottom 2 bytes of the return address is overwritten by something inside LoadAniIcon(). We no longer will return to code in LoadCursorIconFromFileMap(). Instead, we go to 0x77D825BA, which is – guess what – the address of that call [ebx+4] instruction. This is proof that sometimes when you can't overwrite an arbitrary 4 bytes of address to gain control, even an arbitrary 2 bytes will work. Now where did those bytes come from, eh? We're looking for a strict 0x25BA, or some value that is later added, subtracted, multiplied, etc to turn into 0x25BA.

It's easy this time, as the 0x25BA is right inside the malformed ANI file. Someone did their homework and studied the addresses pretty well. I'm going to re-use the same screen shot as the one above…can you spot the bytes?



Yep, they are the first two bytes of what we labeled as the shellcode before. At some point in LoadAniIcon(), there is a copy or move function that allows these bytes to overwrite some of the return address. Now look up in the screen shot where the image is almost cut off (around 0x54 offset) and you will see "52 00 00 00" – which is right after the "anih" header signature. This "52 00 00 00" is the size of the "anih" header chunk. This value is 0x52 really or decimal 82 bytes. If you count, this 82 bytes starts right after the size itself, which is at "30 31 32 33" and goes all the way to include the first two bytes of the highlighted area (0xBA25).

So the vulnerability occurs because there is a statically-sized buffer on the stack to hold an "anih" chunk, but it takes the size of the "anih" chunk from the malformed ANI file itself, and doesn't check to see if the specified size will fit. The malformed ANI we know about says that the chunk size is 0x52 bytes and this is too big. The extra bytes overwrite part of the LoadAniIcon() return address and force execution to a location with user32.dll that calls [ebx+4]. Conveniently, [ebx+4] points to the start of shellcode contained within mm.jpg (everything after the 0xBA25, start is highlighted below):



The fix for this is pretty easy, but not easy to detect. We’re basically looking for some DWORD-sized value after an “anih” header that is larger than the buffer on the stack. All for now.