Microsoft's Rich Signature (undocumented)

11/11/2010: corrected the part about the meaning of the @comp.id value high bits.

In this article I'm going to try to provide documentation for the undocumented(*) Rich Signature produced by Microsoft compilers. I'm not completely sure when this signature was introduced, I wrongly believed that I had been introduced with Visual Studio 2003, but I was shown that it is present even in VC++ 6 executables. So, I guess this signature has been introduced with that compiler. Information about this topic is non-existent (seems strange, but it's a fact). Thus, most readers probably don't know what I'm talking about. Let's look at the inital bytes of a normal executable compiled with Visual Studio 2005:

```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.......ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 F8 00 00 00  ............ø...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  º.´.Í!¸LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is.program.canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t.be.run.in.DOS.
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  E7 B3 9D E7 A3 D2 F3 B4 A3 D2 F3 B4 A3 D2 F3 B4  ç³ç£Òó´£Òó´£Òó´
00000090  60 DD AC B4 A8 D2 F3 B4 60 DD AE B4 BE D2 F3 B4  `Ý¬´¨Òó´`Ý®´¾Òó´
000000A0  A3 D2 F2 B4 F8 D0 F3 B4 84 14 8E B4 BA D2 F3 B4  £Òò´øÐó´„Ž´ºÒó´
000000B0  84 14 9E B4 3A D2 F3 B4 84 14 9D B4 3F D2 F3 B4  „ž´:Òó´„´?Òó´
000000C0  84 14 81 B4 B3 D2 F3 B4 84 14 8F B4 A2 D2 F3 B4  „´³Òó´„´¢Òó´
000000D0  84 14 8B B4 A2 D2 F3 B4 52 69 63 68 A3 D2 F3 B4  „‹´¢Òó´Rich£Òó´
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00  ........PE..L.
00000100  01 2F 9A 46 00 00 00 00 00 00 00 00 E0 00 03 01  /šF........à.
00000110  0B 01 08 00 00 10 05 00 00 F0 04 00 00 00 00 00  ....ð.....
00000120  A5 A1 03 00 00 10 00 00 00 20 05 00 00 00 40 00  ¥¡.........@.
00000130  00 10 00 00 00 10 00 00 04 00 00 00 00 00 00 00  .............
00000140  04 00 00 00 00 00 00 00 00 50 0A 00 00 10 00 00  ........P.....
00000150  D8 4B 07 00 02 00 00 00 00 00 10 00 00 10 00 00  ØK.........
```

Ok, we can easily identify the Dos Header followed by the PE Header (also known as NT Headers). But do you also notice something unusual? I'm referring to this:

```
00000070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.......
00000080 E7 B3 9D E7 A3 D2 F3 B4 A3 D2 F3 B4 A3 D2 F3 B4 ç³ç£Òó´£Òó´£Òó´
00000090 60 DD AC B4 A8 D2 F3 B4 60 DD AE B4 BE D2 F3 B4 `Ý¬´¨Òó´`Ý®´¾Òó´
000000A0 A3 D2 F2 B4 F8 D0 F3 B4 84 14 8E B4 BA D2 F3 B4 £Òò´øÐó´„ Ž´ºÒó´
000000B0 84 14 9E B4 3A D2 F3 B4 84 14 9D B4 3F D2 F3 B4 „ ž´:Òó´„ ´?Òó´
000000C0 84 14 81 B4 B3 D2 F3 B4 84 14 8F B4 A2 D2 F3 B4 „ ´³Òó´„ ´¢Òó´
000000D0 84 14 8B B4 A2 D2 F3 B4 52 69 63 68 A3 D2 F3 B4 „ ‹´¢Òó´Rich£Òó´
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000F0 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00 ........PE..L .
```

This seems pretty strange, doesn't it? What's this data between the dos stub and the PE Header?

As you can see, the highlighted block of data contains the word Rich. This why it is called Rich Signature, it surely isn't its internal official name. Not that any serious paper has ever mentioned it as Rich Signature, but in a couple of forums it came up with this name. That's all what is known about this block of data. That it contains the word "Rich" and that it is stored between the Dos Header and the PE Header. Also, what should be noted, is that this signature is only present in VC++ executables, it can't be found in .NET assemblies.

Since I had no information at all available, I started analyzing the data itself. After a bit of time I started noticing something interesting. A costant pattern of identic dwords in the block. Also, other dwords had similarities with the identical dwords. I marked the identical dwords with red and the similarities with yellow.

```
00000080  E7 B3 9D E7 A3 D2 F3 B4 A3 D2 F3 B4 A3 D2 F3 B4   ç³ç£Òó´£Òó´£Òó´
00000090  60 DD AC B4 A8 D2 F3 B4 60 DD AE B4 BE D2 F3 B4   `Ý¬´¨Òó´`Ý®´¾Òó´
000000A0  A3 D2 F2 B4 F8 D0 F3 B4 84 14 8E B4 BA D2 F3 B4   £Òò´øÐó´„Ž´ºÒó´
000000B0  84 14 9E B4 3A D2 F3 B4 84 14 9D B4 3F D2 F3 B4   „ž´:Òó´„´?Òó´
000000C0  84 14 81 B4 B3 D2 F3 B4 84 14 8F B4 A2 D2 F3 B4   „´³Òó´„´¢Òó´
000000D0  84 14 8B B4 A2 D2 F3 B4 52 69 63 68 A3 D2 F3 B4   „‹´¢Òó´Rich£Òó´
```

What made me thinking was that dword following the word Rich. Why put this kind of dword after the word "Rich"? Anyway, I xored the block of data from the beginning to the word "Rich" with this dword and got a very convincing result:

```
00000080 44 61 6E 53 00 00 00 00 00 00 00 00 00 00 00 00 DanS............
00000090 C3 0F 5F 00 0B 00 00 00 C3 0F 5D 00 1D 00 00 00 Ã _. ...Ã ]. ...
000000A0 00 00 01 00 5B 02 00 00 27 C6 7D 00 19 00 00 00 .. .[ ..'Æ}. ...
000000B0 27 C6 6D 00 99 00 00 00 27 C6 6E 00 9C 00 00 00 'Æm.™...'Æn.œ...
000000C0 27 C6 72 00 10 00 00 00 27 C6 7C 00 01 00 00 00 'Ær. ...'Æ|. ...
000000D0 27 C6 78 00 01 00 00 00 52 69 63 68             'Æx. ...Rich
```

This data seems very consistent. Also because it starts with the dword DanS. In fact, the data seemed to me such consistent that I wrote a CFF Explorer script to decrypt further signatures. Don't worry, my intuition was right and the following script is 100% correct.

- Dowload the Rich Signature Decrypter script

```
-- Rich Signature Decrypter
-- © 2008 Daniel Pistelli

filename = GetOpenFile()

if filename == null then
   return
end

hFile = OpenFile(filename)

if hFile == null then
   return
end

nSignDwords = 0

for i = 0, 100 do

   dw = ReadDword(hFile, 0x80 + (i * 4))

   if dw == null then
      return
   end

   -- is this the "Rich" terminator?

   if dw == 0x68636952 then

      nSignDwords = i

      break
   end
end

if nSignDwords == 0 then
   return
end

-- read xor mask
```

```
mask = ReadDword(hFile, 0x80 + ((nSignDwords + 1) * 4))

-- decrypt signature

for i = 0, nSignDwords - 1 do

   dw = ReadDword(hFile, 0x80 + (i * 4))

   WriteDword(hFile, 0x80 + (i * 4), dw ^ mask)
end

-- write new mask for decrypted signature

WriteDword(hFile, 0x80 + ((nSignDwords + 1) * 4), 0xFFFFFFFF)

-- save file

if SaveFile(hFile) == true then
   OpenWithCFFExplorer(filename)
end
```

It's not that I'm trying to spam around the CFF Explorer scripting language (which basically is a modified Lua with 0-based arrays, in case you're not familiar with it), but it's time saving to me and I use it whenever possible: writing the same thing in C/C++ or any other language would take me longer. As I said, this script decrypts every Rich Signature, corrects the xor mask (the magic dword which follows the "Rich" word) and opens the patched file in the CFF Explorer.

My guess was that this signature was produced by the linker, which is, of course, the most obvious choice. So, I disassembled the linker and looked for the "DanS" word. I found it at once. The function which creates the signature is: CbBuildProdidBlock. What follows is the disassembly of this function and all the comments I added to the code. After the disassembly, I'll sum up everything this function does and add further explanation about some points.

```
.text:004650E0 ; unsigned int __stdcall IMAGE__CbBuildProdidBlock(unsigned int Arg1, int Arg2)
.text:004650E0 ?CbBuildProdidBlock@IMAGE@@AAEKPAPAX@Z proc near
.text:004650E0                     ; CODE XREF: BuildImage+1057p
.text:004650E0
.text:004650E0 var_14 = dword ptr -14h
.text:004650E0 var_4 = dword ptr -4
.text:004650E0 Arg1  = dword ptr  4
.text:004650E0 Arg2  = dword ptr  8
.text:004650E0
.text:004650E0      mov eax, ?hheap@Heap@@2PAXA ; void * Heap::hheap
.text:004650E5      sub esp, 14h
.text:004650E8      push ebx
.text:004650E9      push ebp
.text:004650EA      push esi
.text:004650EB      push edi
.text:004650EC      push 0Ch             ; dwBytes: 12
.text:004650EE      push 0               ; dwFlags
.text:004650F0      push eax             ; hHeap
.text:004650F1      call ds:__imp__HeapAlloc@12 ; allocates the space for the first (last)
.text:004650F1                     ; item of the linked list built to create the
.text:004650F1                     ; signature
.text:004650F7      test eax, eax
.text:004650F9      jnz short MemoryOk
.text:004650FB
.text:004650FB FatalError :                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+BEj
.text:004650FB      push 44Eh            ; unsigned int
.text:00465100      push 0               ; unsigned __int16 *
.text:00465102      call ?Fatal@@YAXPBGIZZ  ; Fatal(ushort const *,uint,...)
.text:00465107 ; ---------------------------------------------------------------------------
.text:00465107
```

```
.text:00465107 MemoryOk :                  ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+19j
.text:00465107      mov edx, [esp+ 24h + Arg1 ]
.text:0046510B      mov ecx, 1
.text:00465110      mov ebp, eax           ; from here a linked list is created
.text:00465110                             ; every item of the list is 12 bytes big:
.text:00465110                             ; 2 dwords of data and a pointer to the next
.text:00465110                             ; element of the list. The structure would be:
.text:00465110                             ; pointer (dword)
.text:00465110                             ; data1 (dword)
.text:00465110                             ; data2 (dword)
.text:00465112      mov dword ptr [eax], 0  ; pointer = 0
.text:00465118      mov dword ptr [eax+ 4 ], 78C627h ; data1 = 78C627h
.text:0046511F      mov [eax+ 8 ], ecx       ; data2 = 1
.text:0046511F                             ; the last dwords of a Rich signature are always
.text:0046511F                             ; 0078C627 and 00000001h
.text:00465122      mov eax, [edx+ 23Ch ]     ; it's a pointer
.text:00465122                             ; we'll call this pointer XS
.text:00465122                             ; (edx contains a pointer to the
.text:00465122                             ; Microsoft Linker Database)
.text:00465128      mov [esp+ 24h + var_14 ], ecx
.text:0046512C      xor edi, edi
.text:0046512E      mov [esp+ 24h + var_4 ], eax
.text:00465132
.text:00465132 loc_465132:                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+76j
.text:00465132      test edi, edi
.text:00465134      jnz short loc_46513C
.text:00465136      mov edi, [esp+ 24h + var_4 ]
.text:0046513A      jmp short loc_46513F
.text:0046513C ; ------------------------------------------------------------------------
.text:0046513C
.text:0046513C loc_46513C:                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+54j
.text:0046513C      mov edi, [edi+ 44h ]
.text:0046513F
.text:0046513F loc_46513F:                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+5Aj
.text:0046513F      test edi, edi
.text:00465141      jz short buildxormask   ; jumps to the xor mask creation
.text:00465143      xor ebx, ebx
.text:00465145
.text:00465145 NextItem :                  ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+D7j
.text:00465145                             ; IMAGE::CbBuildProdidBlock(void * *)+DCj
.text:00465145      test ebx, ebx          ; ebx == 0 ?
.text:00465147      jnz short loc_46514E    ; ebx is 0 only in the first time the loop is executed
.text:00465149      mov ebx, [edi+ 40h ]      ; ebx = [XS+40h] (only first execution)
.text:00465149                             ; now ebx contains another pointer
.text:00465149                             ; let's call this pointer YS
.text:0046514C      jmp short loc_465154
.text:0046514E ; ------------------------------------------------------------------------
.text:0046514E
.text:0046514E loc_46514E:                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+67j
.text:0046514E      mov ebx, [ebx+ 0A0h ]     ; ebx = YS pointer
.text:00465154
.text:00465154 loc_465154:                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+6Cj
.text:00465154      test ebx, ebx
.text:00465156      jz short loc_465132
.text:00465158      mov esi, [ebx+ 74h ]      ; esi = [YS+74h] (new data1)
.text:0046515B      test esi, 0FFFF0000h    ; high word is zero?
.text:00465161      jnz short loc_465179    ; if not, jump to new alloc
.text:00465163      mov dl, [ebx+ 0ACh ]
.text:00465169      shr dl, 6
.text:0046516C      test dl, 1
.text:0046516F      jz short loc_465179
.text:00465171      mov eax, ?g_pmodCIL@@3PAUMOD@@A ; MOD * g_pmodCIL
.text:00465176      mov esi, [eax+ 74h ]      ; seems to be a fixed value
.text:00465176                             ; esi = new data1 (fixed)
.text:00465179
```

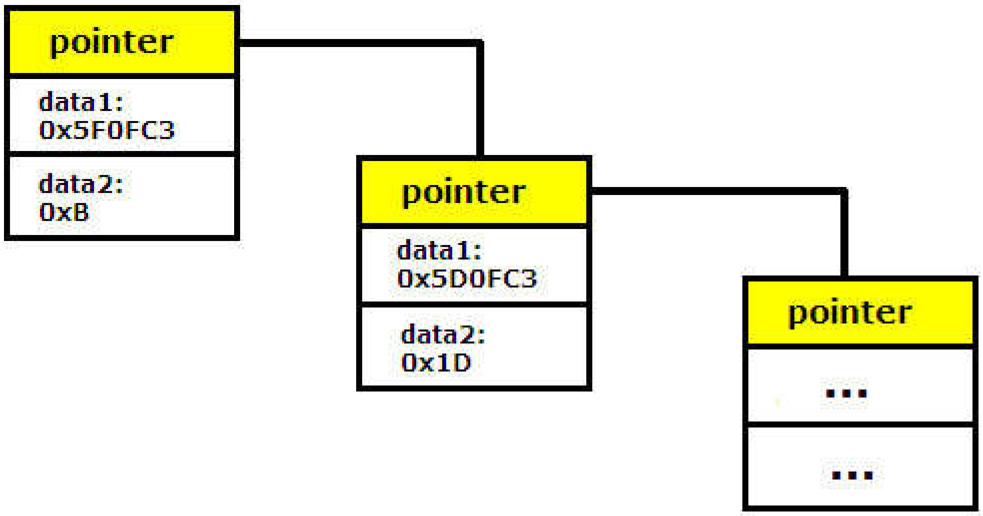
```
.text:00465179 loc_465179:                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+81j
.text:00465179                         ; IMAGE::CbBuildProdidBlock(void * *)+8Fj
.text:00465179       test ebp, ebp
.text:0046517B       mov eax, ebp
.text:0046517D       jz short loc_46518B
.text:0046517F       nop
.text:00465180
.text:00465180 loc_465180:                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+A9j
.text:00465180       cmp [eax+ 4 ], esi        ; old data1 == new data1 ?
.text:00465183       jz short IncrementData2
.text:00465185       mov eax, [eax]          ; eax = linked list pointer
.text:00465187       test eax, eax           ; is the pointer to the next item null?
.text:00465189       jnz short loc_465180    ; if not, repeats the check
.text:00465189                         ; this basically checks if one of the
.text:00465189                         ; other data1 elements in the list
.text:00465189                         ; equals the new data1 value
.text:00465189                         ;
.text:00465189                         ; if none of the items is equal,
.text:00465189                         ; it allocates the memory for the new item in the list
.text:00465189                         ;
.text:00465189                         ; if an old item data1 equals the new data1,
.text:00465189                         ; then the data2 dword of that old item is incremented
.text:0046518B
.text:0046518B loc_46518B:                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+9Dj
.text:0046518B       mov ecx, ?hheap@Heap@@2PAXA ; void * Heap::hheap
.text:00465191       push 0Ch               ; dwBytes (the usual 12 bytes)
.text:00465193       push 0                ; dwFlags
.text:00465195       push ecx               ; hHeap
.text:00465196       call ds: __imp__HeapAlloc@12 ; HeapAlloc(x,x,x)
.text:0046519C       test eax, eax
.text:0046519E       jz FatalError
.text:004651A4       mov ecx, 1
.text:004651A9       add [esp+ 24h + var_14 ], ecx ; increments the linked list count dword
.text:004651AD       mov [eax], ebp          ; new item pointer
.text:004651AF       mov [eax+ 4 ], esi        ; new data1
.text:004651B2       mov [eax+ 8 ], ecx        ; data2 = 1 (for now)
.text:004651B5       mov ebp, eax
.text:004651B7       jmp short NextItem
.text:004651B9 ; ---------------------------------------------------------------------------
.text:004651B9
.text:004651B9 IncrementData2 :             ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+A3j
.text:004651B9       add [eax+ 8 ], ecx        ; increment data2
.text:004651BC       jmp short NextItem
.text:004651BE ; ---------------------------------------------------------------------------
.text:004651BE
.text:004651BE buildxormask :               ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+61j
.text:004651BE       mov ecx, [esp+ 24h + Arg1 ]
.text:004651C2       mov edx, [ecx+ 25Ch ]     ; initial xor mask value (0x80)
.text:004651C2                         ; 0x80 is also the value of the
.text:004651C2                         ; loops in the first xor mask loop
.text:004651C2                         ; (remember 0x80 is also the offset
.text:004651C2                         ; where the Rich Signature is stored)
.text:004651C8       xor eax, eax
.text:004651CA       test edx, edx           ; is initial mask (counter) != 0 ?
.text:004651CC       mov esi, edx
.text:004651CE       jbe short xoormaskloop2cond ; if so, skip the first xor mask loop
.text:004651D0       mov ebx, [ecx+ 258h ]     ; ebx = pointer to the linked executable
.text:004651D0                         ; I'll call this pointer "PointerToPE"
.text:004651D0                         ; the following loop basically creates the xor mask from
.text:004651D0                         ; a checksum of the first bytes of our executable
.text:004651D6
.text:004651D6 xormaskloop1 :               ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+105j
.text:004651D6       movzx edi, byte ptr [ebx+eax] ; edi = (BYTE) PointerToPE[eax]
.text:004651DA       mov cl, al              ; low byte of loop counter in cl
.text:004651DC       rol edi, cl             ; rotates left the current byte of PointerToPE
```

```
.text:004651DC                          ; with the low byte of the loop counter
.text:004651DE      add eax, 1              ; increment eax
.text:004651E1      add esi, edi            ; adds the result of the rol to the xor mask
.text:004651E3      cmp eax, edx            ; is counter < initial xor mask value?
.text:004651E5      jb short xormaskloop1   ; if so, goes on with the loop
.text:004651E7
.text:004651E7 xoormaskloop2cond :            ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+EEj
.text:004651E7      test ebp, ebp           ; if the initial ebp value is 0,
.text:004651E7                          ; it doesn't execute the second xor mask loop
.text:004651E9      mov eax, ebp            ; eax = initial value for the loop
.text:004651EB      jz short alloc_sign_memory
.text:004651ED      lea ecx, [ecx+ 0 ]        ; ecx = first linked list item
.text:004651ED                          ; the second xor mask loop adds the a checksum
.text:004651ED                          ; of the linked list items to the already existing
.text:004651ED                          ; xor mask value
.text:004651F0
.text:004651F0 xormaskloop2 :                 ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+11Ej
.text:004651F0      mov edx, [eax+ 4 ]        ; edx = data1
.text:004651F3      mov cl, [eax+ 8 ]         ; cl = (BYTE) data2
.text:004651F6      mov eax, [eax]          ; eax = next list item
.text:004651F8      rol edx, cl             ; rotates left edx with cl
.text:004651FA      add esi, edx            ; adds the result of the rol to the xor mask
.text:004651FC      test eax, eax           ; pointer = 0?
.text:004651FE      jnz short xormaskloop2  ; if not, goes on with the loop
.text:00465200
.text:00465200 alloc_sign_memory :            ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+10Bj
.text:00465200      xor edx, edx
.text:00465202      mov eax, esi            ; puts the xor mask in eax
.text:00465204      shr eax, 5              ; shifts the xor mask 5 bits right
.text:00465207      mov ecx, 3
.text:0046520C      div ecx                 ; divides the result by 3
.text:0046520E      add edx, [esp+ 24h + var_14 ] ; adds the mod of the division to the
.text:0046520E                          ; count of the linked list items
.text:0046520E                          ; this means the number of data1 and data2
.text:0046520E                          ; in the linked list item
.text:00465212      lea ebx, ds: 20h [edx*8]  ; multiplies the whole thing and adds 20h
.text:00465212                          ; ebx now contains the size of the rich signature
.text:00465212                          ; what just happened might appear strange and
.text:00465212                          ; I'll explain later the meaning of this whole
.text:00465212                          ; operation since it's quite interesting
.text:00465219      mov edx, ?hheap@Heap@@2PAXA ; void * Heap::hheap
.text:0046521F      push ebx                ; dwBytes
.text:00465220      push 8                  ; dwFlags
.text:00465222      push edx                ; hHeap
.text:00465223      mov [esp+ 30h + Arg1 ], ebx
.text:00465227      call ds: __imp__HeapAlloc@12 ; HeapAlloc(x,x,x)
.text:0046522D      test eax, eax
.text:0046522F      jnz short buildsign
.text:00465231      push 44Eh               ; unsigned int
.text:00465236      push eax                ; unsigned __int16 *
.text:00465237      call ?Fatal@@YAXPBGIZZ  ; Fatal(ushort const *,uint,...)
.text:0046523C ; -----------------------------------------------------------------------
.text:0046523C
.text:0046523C buildsign :                    ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+14Fj
.text:0046523C      mov ecx, [esp+ 24h + Arg2 ]
.text:00465240      mov [ecx], eax          ; stores the ptr of the allocated memory
.text:00465240                          ; for the signature at the address
.text:00465240                          ; specified by the function's second argument
.text:00465242      mov edx, esi            ; moves the xor mask into edx
.text:00465244      lea edi, [eax+ 8 ]
.text:00465247      xor edx, 536E6144h      ; xors the "DanS" Dword with the xor mask
.text:0046524D      mov [eax], edx          ; stores the "DanS" word into the signature
.text:0046524F      mov [eax+ 4 ], esi        ; stores the xor mask value three times
.text:00465252      mov [edi], esi
.text:00465254      mov [edi+ 4 ], esi
```

```
.text:00465257    add edi, 8           ; increments the signature pointer
.text:0046525A    test ebp, ebp
.text:0046525C    mov eax, ebp
.text:0046525E    jz short NoMoreData
.text:00465260    mov ebp, ds: __imp__HeapFree@12 ; HeapFree(x,x,x)
.text:00465266
.text:00465266 WriteSignatureDwords :       ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+1A9j
.text:00465266    mov ecx, [eax+ 4 ]      ; gets data1
.text:00465269    xor ecx, esi          ; xors it with the mask
.text:0046526B    mov [edi], ecx        ; stores it in the signature
.text:0046526D    mov edx, [eax+ 8 ]      ; gets data2
.text:00465270    xor edx, esi          ; xors it with the mask
.text:00465272    push eax
.text:00465273    mov [edi+ 4 ], edx      ; stores it in the signature
.text:00465276    mov ebx, [eax]
.text:00465278    mov eax, ?hheap@Heap@@2PAXA ; void * Heap::hheap
.text:0046527D    push 0               ; dwFlags
.text:0046527F    push eax             ; hHeap
.text:00465280    add edi, 8
.text:00465283    call ebp ; HeapFree(x,x,x) ;  = current item in the
.text:00465283                        ; list is being freed
.text:00465285    test ebx, ebx         ; next element pointer == 0?
.text:00465287    mov eax, ebx
.text:00465289    jnz short WriteSignatureDwords ; (this loop stores all the
.text:00465289                        ; "secret" data into the signature)
.text:00465289                        ; if the current item pointer is == 0,
.text:00465289                        ; stop the loop
.text:0046528B    mov ebx, [esp+ 24h + Arg1 ]
.text:0046528F
.text:0046528F NoMoreData :                ; CODE XREF: IMAGE::CbBuildProdidBlock(void * *)+17Ej
.text:0046528F    mov [edi+ 4 ], esi      ; stores the xor mask at the end of the signature
.text:00465292    mov dword ptr [edi], 68636952h ; stores the "Rich" Dword at the end
.text:00465292                        ; of the crypted data
.text:00465292                        ; just before the xor mask
.text:00465298    pop edi
.text:00465299    pop esi
.text:0046529A    pop ebp
.text:0046529B    mov eax, ebx          ; ebx contains the size of the rich signature
.text:0046529B                        ; this is the return value of the function
.text:0046529D    pop ebx
.text:0046529E    add esp, 14h
.text:004652A1    retn 8
.text:004652A1 ?CbBuildProdidBlock@IMAGE@@AAEKPAPAX@Z endp
```

The comments in the code give a general idea, but much more clarification is needed. Before I explain anything, let me tell you that if you have in mind to debug the linker, you'll need a kernel debugger: you absolutely cannot debug the linker with ollydbg or other user mode debuggers (don't take this too literally, in theory it's possible, but I being very old school was unable to do so). I did most of the code analyzing just with the disassembler, but at some point I needed to see the values of some variables and in some cases the memory pointed to by these values. So, it took me about a day (I had some troubles) setting up a VM with SoftIce and Visual Studio, which was the best choice in this case, because I needed to set a system breakpoint to attach the debugger to the linker process which is executed by the Visual Studio IDE when building a solution. Another thing: this code is the same as for x64 executables and itanium ones. So, what I'm disclosing here is valid for other platforms as well.

Brief summary of what this function does. The first part of the function creates a linked list of structures containing (not counting the linking pointer) two dwords which I called "data1" and "data2". This list contains one fixed item (data1=0x78C627 and data2=1) which is going to be the last item of the list. Other items are retrieved through a not so clear loop block from the "Microsoft Linker Database". This is simplified, of course, I have analyzed the loop and have a quite good knowledge of what it does, but it's not interesting for us right now, since we don't know the nature of the data. What should be noted in this loop, is that the data2 dword of one linked list item is being incremented everytime the dword data1 of the same item is encountered by the loop walking through the linker's internal data structures.



The linked list is, of course, terminated by the first null pointer. Keep in mind that the last item of this list has a fixed value of data1=0x78C627 and data2=1. After having created the linked list, the function creates the xor mask for the signature. This value should interest us for many reasons, but mainly because we couldn't know before analyzing the function what data the xor mask was made of. As I discovered dissassembling the code, the xor mask is completely harmless, since it's just a result of two checksums. The first checksum is made on the first 0x80 bytes of our PE (remember that the Rich Signature is stored exactly after those first 0x80 bytes), meanwhile the second one is made on the linked list data1 and data2 values. Also, the initial value of the xor mask is always 0x80. So, what information can be retrieved from the xor mask? Only that our Rich Signature or the first 0x80 bytes of our PE haven't been modified. The final part of the function allocates the memory for the signature (I'll talk about that later). Then the dword "DanS" is xored and stored at the beginning of the newly allocated memory. What follows is three times the xor mask value. Then a loop is started which walks through the linked list and xors every data1 and data2 values and stores them into the signature. When the loop is over the dword "Rich" is stored followed by the xor mask. The function returns the size of the new signature and the pointer of the newly allocated memory for the signature is stored through an argument pointer of the function.

This is all pretty clear, the only thing that should be discussed further is the memory allocation process for the signature. This process will decide the effective size of the signature. Let's look at this signature:

```
00000070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.......
00000080 44 61 6E 53 00 00 00 00 00 00 00 00 00 00 00 00 DanS............
00000090 C3 0F 5F 00 0B 00 00 00 C3 0F 5D 00 1D 00 00 00 Ã _. ...Ã ]. ...
000000A0 00 00 01 00 5B 02 00 00 27 C6 7D 00 19 00 00 00 .. .[ ..'Æ}. ...
000000B0 27 C6 6D 00 99 00 00 00 27 C6 6E 00 9C 00 00 00 'Æm.™...'Æn.œ...
000000C0 27 C6 72 00 10 00 00 00 27 C6 7C 00 01 00 00 00 'Ær. ...'Æ|. ...
000000D0 27 C6 78 00 01 00 00 00 52 69 63 68 FF FF FF FF 'Æx. ...Richÿÿÿÿ
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000F0 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00 ........PE..L .
```

The yellow marked data (well, not the entire yellow block, but we'll see that later) between our signature and the PE header is random padding produced by the memory allocation process for the signature. If we consider the code:

```
.text:00465200    xor edx, edx
.text:00465202    mov eax, esi        ; puts the xor mask in eax
.text:00465204    shr eax, 5          ; shifts the xor mask 5 bits right
.text:00465207    mov ecx, 3
.text:0046520C    div ecx             ; divides the result by 3
```

```
.text:0046520E      add edx, [esp+ 24h + var_14 ] ; adds the mod of the division to the
.text:0046520E                          ; count of the linked list items
.text:0046520E                          ; this means the number of data1 and data2
.text:0046520E                          ; in the linked list item
.text:00465212      lea ebx, ds: 20h [edx*8]  ; multiplies the whole thing and adds 20h
```

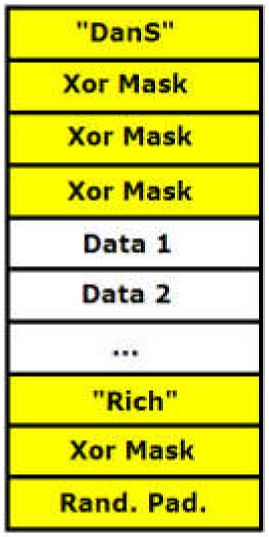
This would look something like this in C/C++:

```
SignSize = ((((XorMask >> 5) % 3) + nListItems) * 8) + 0x20;
```

Since the mod of the xor mask is added to the number of items in the linked list and then multiplied by 8 (size of data1 + data2), the random padding will always be a multiple of 8; and it can be either 0, 8 or 16 bytes. This seems not to be the case in the signature above (the yellow marked data is 24 bytes. But this can be easily explained if you consider the addition of the 0x20 value. To give a clear idea of what I mean, here's the data marked with three different colors:

```
00000080 44 61 6E 53 00 00 00 00 00 00 00 00 00 00 00 00 DanS............
00000090 C3 0F 5F 00 0B 00 00 00 C3 0F 5D 00 1D 00 00 00 Ã _. ...Ã ]. ...
000000A0 00 00 01 00 5B 02 00 00 27 C6 7D 00 19 00 00 00 .. .[ ..'Æ}. ...
000000B0 27 C6 6D 00 99 00 00 00 27 C6 6E 00 9C 00 00 00 'Æm.™...'Æn.œ...
000000C0 27 C6 72 00 10 00 00 00 27 C6 7C 00 01 00 00 00 'Ær. ...'Æ|. ...
000000D0 27 C6 78 00 01 00 00 00 52 69 63 68 FF FF FF FF 'Æx. ...Richÿÿÿÿ
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000F0 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00 ........PE..L .
```

The data in red represents the 0x20 value, the turquoise data is our linked list and the yellow data is the random padding. So, to sum up here's our Rich Signature structure diagram:



Still, we don't know yet what the linked list data is. To discover the nature of the linked list data we have to go back to the linked list's generation loop. The initial memory value from what it all starts is stored here:

```
.text:00465122      mov eax, [edx+ 23Ch ]      ; it's a pointer
.text:00465122                          ; we'll call this pointer XS
.text:00465122                          ; (edx contains a pointer to the
.text:00465122                          ; Microsoft Linker Database)
```

As stated in the comments, edx contains a pointer to the Microsoft Linker Database, which is a structure in memory which starts with this string (so that's what it is, I guess). Since 23Ch is a pretty uncommon number, I looked for other references in the disassembly for that number and collected some interesting findings. Most of the time what I encountered was something like this:

```
.text:00456DC9
.text:00456DC9 loc_456DC9:                ; CODE XREF: FreeImage(IMAGE * *,bool)+7j
.text:00456DC9      mov eax, [ebx]
.text:00456DCB      add eax, 23Ch
.text:00456DD0      push eax              ; struct LIBS *
.text:00456DD1      call ?FreePLIB@@YGXPAULIBS@@@Z ; FreePLIB(LIBS *)
```

So, my guess is that the pointer referenced in our loop was the "struct LIBS *" kind. I admit this was only a guess, but the data in memory couldn't tell me very much and this kind of code references seemed to me solid enough to give this theory a try.

```
struct MicrosoftLinkerDB
{
    BYTE Pad[0x23C]; // unknown data

    struct LIBS *pLibs; // our pointer
};
```

This would be our current structure. I analyzed the linked list loop long enough to have an idea of how the LIBS struct is represented in memory, but I won't disclose it here, because it doesn't add anything to what is necessary to know for what I'm going to do now. As I said, I started from the theory that somehow libraries were involved. So, I took one data1 element from our linked list and searched it in all library files in my SDK. Since I didn't have a tool to search for hex bytes in all files in a directory (all tools I have on my computer work with strings, not bytes), I wrote a little CFF Explorer script to accomplish this task.

- Download Search Hex script

```
-- data to search
data = { 0xC3, 0x0F, 0x5F, 0x00 }

str = GetDirectory("Select Directory...")

n = 0

if str then
   hSearchHandle = InitFindFile(str .. "\\*.*")

   if hSearchHandle then
      FName = FindFile(hSearchHandle)

      while FName do

         off = SearchBytes(str .. "\\" .. FName, 0, data)

         if off != null then
            n = n + 1
            -- print filename
            MsgBox(FName)
         end

         FName = FindFile(hSearchHandle)
      end
   end
end

MsgBox("Number of results: " .. n)
```

The result that I got searching for the value 0x005F0FC3 where:

ADSIid.lib
Bits.lib
bufferoverflowu.lib

certidl.lib
ComMode.obj
Fci.lib
Fdi.lib
GlAux.lib
ksuser.lib
MMC.lib
MqOA.lib
MsXml2.lib
ScrnSave.lib
ScrnSavW.lib
Shell32.lib
strsafe.lib
Svcguid.lib
unicows.lib
Uuid.lib
WbemUuid.lib
WiaGuid.lib
WinStrm.lib
WS2_32.lib

Number of results: 23

If I had suspected such a long list, I would have used the log functions of the scripting. Anyway, I took one of these libraries and opened with the CFF Explorer hex editor. I chose Shell32, because I knew I had included it in my project. So, I opened the library and searched for the value. This is where I found it:

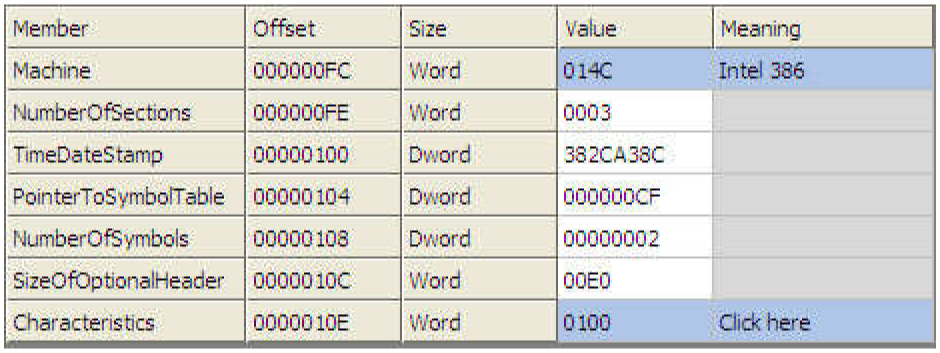
```
0000FFC0 12 4D 69 63 72 6F 73 6F 66 74 20 28 52 29 20 4C Microsoft.(R).L
0000FFD0 49 4E 4B 00 00 00 00 00 00 00 00 00 40 63 6F 6D INK.........@com
0000FFE0 70 2E 69 64 C3 0F 5D 00 FF FF 00 00 03 00 00 00 p.idÃ ].ÿÿ.. ...
0000FFF0 00 00 04 00 00 00 00 00 00 00 02 00 00 00 02 00 .. ....... ... .
```

As you can see, our value comes right after the string "@comp.id". And this is true for every library: I checked every data1 value of the linked list. Every value could be found in one or more input libraries used by our test executable, and those data1 values were always preceded by this string in the libraries. One library can have many different "@comp.id" values, meanwhile object files have only one. To learn why this is so, we have to take a look at the format of both libraries and object files. Well, to tell the truth, it's the same format: a library contains one or more object files. Before I say anything, you should know that the library and the object file formats aren't as popular as the PE format, so there is less written material about them. Telling a long story short, a library starts with a 8 byte string, followed by a IMAGE_ARCHIVE_MEMBER_HEADER structure. After this structure comes a dword (in big endian, most values in a library are in big endian format) which tells us the number of symbols contained in the library. This dword is followed by an array of big endian dwords (the size was given by the symbols' number value) with the offsets of the symbols' data and after this array comes an array of zero terminated strings, one for each symbol. Data associated with the symbol can be IMAGE_ARCHIVE_MEMBER_HEADER followed by an object file. This brings us to the format of an object file. This is very simple, it's just a File Header structure followed by a Section Headers array. In the hex data below two object files are visible. I marked the Archive Member Header structure in gray, the File Header in yellow and the Sections Header array in red. Notice that I marked the "@comp.id" string in orange.

```
00000790 43 48 45 44 32 30 2E 64 6C 6C 2F 20 20 20 39 34 CHED20.dll/...94
000007A0 32 34 34 39 35 34 38 20 20 20 20 20 20 20 20 20 2449548.........
000007B0 20 20 20 20 20 20 30 20 20 20 20 20 20 20 32 34 ......0.......24
000007C0 34 20 20 20 20 20 20 20 60 0A 4C 01 02 00 8C A3 4.......`.L .Œ£
000007D0 2C 38 B3 00 00 00 02 00 00 00 00 00 00 01 2E 64 ,8³... ...... .d
000007E0 65 62 75 67 24 53 00 00 00 00 00 00 00 00 3B 00 ebug$S........;.
000007F0 00 00 64 00 00 00 00 00 00 00 00 00 00 00 00 00 ..d.............
00000800 00 00 40 00 10 42 2E 69 64 61 74 61 24 33 00 00 ..@. B.idata$3..
00000810 00 00 00 00 00 00 14 00 00 00 9F 00 00 00 00 00 ...... ...Ÿ.....
```

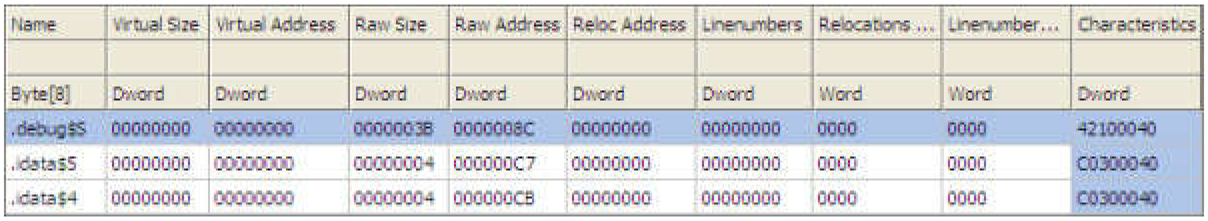
```
00000820 00 00 00 00 00 00 00 00 00 00 40 00 30 C0 01 00 ..........@.0À .
00000830 00 00 13 00 09 00 00 00 00 00 0C 52 49 43 48 45 .. ....... RICHE
00000840 44 32 30 2E 64 6C 6C 20 00 01 00 03 07 00 08 18 D20.dll.. . .
00000850 4D 69 63 72 6F 73 6F 66 74 20 4C 49 4E 4B 20 35 Microsoft.LINK.5
00000860 2E 31 32 2E 39 30 34 39 00 00 00 00 00 00 00 00 .12.9049........
00000870 00 00 00 00 00 00 00 00 00 00 00 00 00 40 63 6F .............@co
00000880 6D 70 2E 69 64 59 23 13 00 FF FF 00 00 03 00 00 mp.idY# .ÿÿ.. ..
00000890 00 00 00 04 00 00 00 00 00 00 00 02 00 00 00 02 ... ....... ...
000008A0 00 1D 00 00 00 5F 5F 4E 55 4C 4C 5F 49 4D 50 4F . ...__NULL_IMPO
000008B0 52 54 5F 44 45 53 43 52 49 50 54 4F 52 00 52 49 RT_DESCRIPTOR.RI
000008C0 43 48 45 44 32 30 2E 64 6C 6C 2F 20 20 20 39 34 CHED20.dll/...94
000008D0 32 34 34 39 35 34 38 20 20 20 20 20 20 20 20 20 2449548.........
000008E0 20 20 20 20 20 20 30 20 20 20 20 20 20 20 32 37 ......0.......27
000008F0 33 20 20 20 20 20 20 60 0A 4C 01 03 00 8C A3 3.......`.L .Œ£
00000900 2C 38 CF 00 00 00 02 00 00 00 00 00 00 01 2E 64 ,8Ï... ...... .d
00000910 65 62 75 67 24 53 00 00 00 00 00 00 00 00 3B 00 ebug$S........;.
00000920 00 00 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 ..Œ.............
00000930 00 00 40 00 10 42 2E 69 64 61 74 61 24 35 00 00 ..@. B.idata$5..
00000940 00 00 00 00 00 00 04 00 00 00 C7 00 00 00 00 00 ...... ...Ç.....
00000950 00 00 00 00 00 00 00 00 00 00 40 00 30 C0 2E 69 ..........@.0À.i
00000960 64 61 74 61 24 34 00 00 00 00 00 00 00 00 04 00 data$4........ .
00000970 00 00 CB 00 00 00 00 00 00 00 00 00 00 00 00 00 ..Ë.............
00000980 00 00 40 00 30 C0 01 00 00 00 13 00 09 00 00 00 ..@.0À ... .....
00000990 00 00 0C 52 49 43 48 45 44 32 30 2E 64 6C 6C 20 .. RICHED20.dll.
000009A0 00 01 00 03 07 00 08 18 4D 69 63 72 6F 73 6F 66 . . . Microsof
000009B0 74 20 4C 49 4E 4B 20 35 2E 31 32 2E 39 30 34 39 t.LINK.5.12.9049
000009C0 00 00 00 00 00 00 00 00 00 40 63 6F 6D 70 2E 69 .........@comp.i
000009D0 64 59 23 13 00 FF FF 00 00 03 00 00 00 00 00 04 dY# .ÿÿ.. .....
000009E0 00 00 00 00 00 00 00 02 00 00 00 02 00 1E 00 00 ....... ... . ..
000009F0 00 7F 52 49 43 48 45 44 32 30 5F 4E 55 4C 4C 5F .RICHED20_NULL_
```

So, as you can see, every object file has only one "@comp.id" string, and a library can have more than one, because it embeds more than just one object file. Before thinking about the nature of the value following the "@comp.id" string, I wanted to know how to get to this value in the first place. To analyze both the File Header and the Sections Header array properly I needed the CFF Explorer. Since the CFF Explorer doesn't support object files (yet), I overwrote the File Header and the Sections Header array of a Portable Executable file. Let's take a look at the data we're analyzing in a simpler way:



Member	Offset	Size	Value	Meaning
Machine	000000FC	Word	014C	Intel 386
NumberOfSections	000000FE	Word	0003	
TimeDateStamp	00000100	Dword	382CA38C	
PointerToSymbolTable	00000104	Dword	000000CF	
NumberOfSymbols	00000108	Dword	00000002	
SizeOfOptionalHeader	0000010C	Word	00E0	
Characteristics	0000010E	Word	0100	Click here

Nevermind the SizeOfOptionalHeader. This field is 0 in an object file. I changed it because it is necessary in a PE file (it tells where the Section Headers array is located). And the sections:



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Linenumber...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.debug$S	00000000	00000000	0000003B	0000008C	00000000	00000000	0000	0000	42100040
.idata$5	00000000	00000000	00000004	000000C7	00000000	00000000	0000	0000	C0300040
.idata$4	00000000	00000000	00000004	000000CB	00000000	00000000	0000	0000	C0300040

What I tried first was checking if the string was contained in a section. The size of a section is given by the Raw Size field, whereas the location is given by the Raw Address. To obtain the file offset of the section, one has to add the File Header offset to the Raw Address field. In this object I marked the data of the last section:

```
000008C0 43 48 45 44 32 30 2E 64 6C 6C 2F 20 20 20 39 34 CHED20.dll/...94
000008D0 32 34 34 39 35 34 38 20 20 20 20 20 20 20 20 20 2449548.........
000008E0 20 20 20 20 20 20 30 20 20 20 20 20 20 20 32 37 ......0.......27
000008F0 33 20 20 20 20 20 20 20 60 0A 4C 01 03 00 8C A3 3.......`.L .Œ£
00000900 2C 38 CF 00 00 00 02 00 00 00 00 00 00 01 2E 64 ,8Ï... ...... .d
00000910 65 62 75 67 24 53 00 00 00 00 00 00 00 00 3B 00 ebug$S........;.
00000920 00 00 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 ..Œ.............
00000930 00 00 40 00 10 42 2E 69 64 61 74 61 24 35 00 00 ..@. B.idata$5..
00000940 00 00 00 00 00 00 04 00 00 00 C7 00 00 00 00 00 ...... ...Ç.....
00000950 00 00 00 00 00 00 00 00 00 00 40 00 30 C0 2E 69 ..........@.0À.i
00000960 64 61 74 61 24 34 00 00 00 00 00 00 00 00 04 00 data$4........ .
00000970 00 00 CB 00 00 00 00 00 00 00 00 00 00 00 00 00 ..Ë.............
00000980 00 00 40 00 30 C0 01 00 00 00 13 00 09 00 00 00 ..@.0À ... .....
00000990 00 00 0C 52 49 43 48 45 44 32 30 2E 64 6C 6C 20 .. RICHED20.dll.
000009A0 00 01 00 03 07 00 08 18 4D 69 63 72 6F 73 6F 66 . . . Microsof
000009B0 74 20 4C 49 4E 4B 20 35 2E 31 32 2E 39 30 34 39 t.LINK.5.12.9049
000009C0 00 00 00 00 00 00 00 00 00 40 63 6F 6D 70 2E 69 .........@comp.i
000009D0 64 59 23 13 00 FF FF 00 00 03 00 00 00 00 00 04 dY# .ÿÿ.. .....
```

As you can see the "@comp.id" string comes right after the last section's data. Our data is not contained in a section as it turned out. I gave another look at the File Header structure and noticed the PointerToSymbolTable field, which in this case is 0xCF. Look at what happens if I mark 0xCF bytes starting from the File Header (which is always the base offset in object files):

```
000008F0 33 20 20 20 20 20 20 20 60 0A 4C 01 03 00 8C A3 3.......`.L .Œ£
00000900 2C 38 CF 00 00 00 02 00 00 00 00 00 00 01 2E 64 ,8Ï... ...... .d
00000910 65 62 75 67 24 53 00 00 00 00 00 00 00 00 3B 00 ebug$S........;.
00000920 00 00 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 ..Œ.............
00000930 00 00 40 00 10 42 2E 69 64 61 74 61 24 35 00 00 ..@. B.idata$5..
00000940 00 00 00 00 00 00 04 00 00 00 C7 00 00 00 00 00 ...... ...Ç.....
00000950 00 00 00 00 00 00 00 00 00 00 40 00 30 C0 2E 69 ..........@.0À.i
00000960 64 61 74 61 24 34 00 00 00 00 00 00 00 00 04 00 data$4........ .
00000970 00 00 CB 00 00 00 00 00 00 00 00 00 00 00 00 00 ..Ë.............
00000980 00 00 40 00 30 C0 01 00 00 00 13 00 09 00 00 00 ..@.0À ... .....
00000990 00 00 0C 52 49 43 48 45 44 32 30 2E 64 6C 6C 20 .. RICHED20.dll.
000009A0 00 01 00 03 07 00 08 18 4D 69 63 72 6F 73 6F 66 . . . Microsof
000009B0 74 20 4C 49 4E 4B 20 35 2E 31 32 2E 39 30 34 39 t.LINK.5.12.9049
000009C0 00 00 00 00 00 00 00 00 00 40 63 6F 6D 70 2E 69 .........@comp.i
000009D0 64 59 23 13 00 FF FF 00 00 03 00 00 00 00 00 04 dY# .ÿÿ.. .....
```

The field PointerToSymbolTable seems to bring us directly to the "@comp.id" string. However, it's not always that easy, sometimes this field brings us slightly before the string. Intrestingly, we can get to the string by multiples of 0x12 bytes, and after the string often come section headers. This seemed to me as a sort of table (and the field PointerToSymbolTable would suggest that I'm right). I remembered a very handy utility shipped along with Microsoft's compilers called dumpbin. I ran it on the library we analyzed above, which, by the way, is the Riched32.lib. Here is an interesting part of the output:

```
Dump of file Riched32.lib

File Type: LIBRARY

Archive member name at 8: /
382CA38C time/date Sat Nov 13 00:32:28 1999
         uid
         gid
       0 mode
     26C size
correct header end

    21 public symbols
```

```
      566 __IMPORT_DESCRIPTOR_RICHED20
      78E __NULL_IMPORT_DESCRIPTOR
      8BE RICHED20_NULL_THUNK_DATA
      AFE _IID_IRichEditOle
      AFE __imp__IID_IRichEditOle
      B6E _IID_IRichEditOleCallback
      B6E __imp__IID_IRichEditOleCallback
      A8A _CreateTextServices@12
      A8A __imp__CreateTextServices@12
      CC0 _IID_ITextServices
      CC0 __imp__IID_ITextServices
      BE6 _IID_ITextHost
      BE6 __imp__IID_ITextHost
      C52 _IID_ITextHost2
      C52 __imp__IID_ITextHost2
      A0C ?REExtendedRegisterClass@@YGHXZ
      A0C __imp_?REExtendedRegisterClass@@YGHXZ
      DA8 _RichEditANSIWndProc@16
      DA8 __imp__RichEditANSIWndProc@16
      D30 _RichEdit10ANSIWndProc@16
      D30 __imp__RichEdit10ANSIWndProc@16

Archive member name at 2B0: /
382CA38C time/date Sat Nov 13 00:32:28 1999
         uid
         gid
       0 mode
     27A size
correct header end

    13 offsets

        1      566
        2      78E
        3      8BE
        4      AFE
        5      B6E
        6      A8A
        7      CC0
        8      BE6
        9      C52
        A      A0C
        B      DA8
        C      D30
        D        0

    21 public symbols

        A ?REExtendedRegisterClass@@YGHXZ
        6 _CreateTextServices@12
        4 _IID_IRichEditOle
        5 _IID_IRichEditOleCallback
        8 _IID_ITextHost
        9 _IID_ITextHost2
        7 _IID_ITextServices
        C _RichEdit10ANSIWndProc@16
        B _RichEditANSIWndProc@16
        1 __IMPORT_DESCRIPTOR_RICHED20
        2 __NULL_IMPORT_DESCRIPTOR
        A __imp_?REExtendedRegisterClass@@YGHXZ
        6 __imp__CreateTextServices@12
        4 __imp__IID_IRichEditOle
        5 __imp__IID_IRichEditOleCallback
        8 __imp__IID_ITextHost
        9 __imp__IID_ITextHost2
        7 __imp__IID_ITextServices
        C __imp__RichEdit10ANSIWndProc@16
        B __imp__RichEditANSIWndProc@16
        3 RICHED20_NULL_THUNK_DATA

Archive member name at 566: RICHED20.dll/
382CA38C time/date Sat Nov 13 00:32:28 1999
         uid
         gid
       0 mode
     1EB size
correct header end

FILE HEADER VALUES
             14C machine (x86)
               3 number of sections
        382CA38C time date stamp Sat Nov 13 00:32:28 1999
             107 file pointer to symbol table
               8 number of symbols
               0 size of optional header
             100 characteristics
                   32 bit word machine

SECTION HEADER #1
.debug$S name
       0 physical address
       0 virtual address
      3B size of raw data
      8C file pointer to raw data (0000008C to 000000C6)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42100040 flags
         Initialized Data
         Discardable
         1 byte align
         Read Only

RAW DATA #1
  00000000: 01 00 00 00 13 00 09 00 00 00 00 00 0C 52 49 43  .............RIC
  00000010: 48 45 44 32 30 2E 64 6C 6C 20 00 01 00 03 07 00  HED20.dll ......
  00000020: 08 18 4D 69 63 72 6F 73 6F 66 74 20 4C 49 4E 4B  ..Microsoft LINK
  00000030: 20 35 2E 31 32 2E 39 30 34 39 00                 5.12.9049.

SECTION HEADER #2
.idata$2 name
       0 physical address
       0 virtual address
      14 size of raw data
      C7 file pointer to raw data (000000C7 to 000000DA)
      DB file pointer to relocation table
       0 file pointer to line numbers
       3 number of relocations
       0 number of line numbers
C0300040 flags
         Initialized Data
         4 byte align
         Read Write

RAW DATA #2
  00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  00000010: 00 00 00 00                                      ....
```

```
RELOCATIONS #2
                                                Symbol    Symbol
Offset    Type              Applied To         Index     Name
--------  ----------------  -----------------  --------  ------
0000000C  DIR32NB                    00000000         3  .idata$6
00000000  DIR32NB                    00000000         4  .idata$4
00000010  DIR32NB                    00000000         5  .idata$5

SECTION HEADER #3
.idata$6 name
       0 physical address
       0 virtual address
       E size of raw data
      F9 file pointer to raw data (000000F9 to 00000106)
      DB file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0200040 flags
         Initialized Data
         2 byte align
         Read Write

RAW DATA #3
  00000000: 52 49 43 48 45 44 32 30 2E 64 6C 6C 00 00        RICHED20.dll..

COFF SYMBOL TABLE
000 00132359 ABS    notype       Static       | @comp.id
001 00000000 SECT2  notype       External     | __IMPORT_DESCRIPTOR_RICHED20
002 C0000040 SECT2  notype       Section      | .idata$2
003 00000000 SECT3  notype       Static       | .idata$6
004 C0000040 UNDEF  notype       Section      | .idata$4
005 C0000040 UNDEF  notype       Section      | .idata$5
006 00000000 UNDEF  notype       External     | __NULL_IMPORT_DESCRIPTOR
007 00000000 UNDEF  notype       External     | RICHED20_NULL_THUNK_DATA
```

Dumpbin classifies our data as part of the COFF SYMBOL TABLE. What I thought was: if it is in the symbol table, it has to be a symbol. Thus, I searched for a symbol structure in the Winnt.h. I assumed that it was impossible that something defined so clearly was not given a definition in that header. Turns out I was right, here's everything we need to interpret the data correctly:

```
//
// Symbol format.
//

typedef struct _IMAGE_SYMBOL {
    union {
        BYTE    ShortName[8];
        struct {
            DWORD   Short;     // if 0, use LongName
            DWORD   Long;      // offset into string table
        } Name;
        DWORD   LongName[2];    // PBYTE [2]
    } N;
    DWORD   Value;
    SHORT   SectionNumber;
    WORD    Type;
    BYTE    StorageClass;
    BYTE    NumberOfAuxSymbols;
} IMAGE_SYMBOL;
typedef IMAGE_SYMBOL UNALIGNED *PIMAGE_SYMBOL;
```

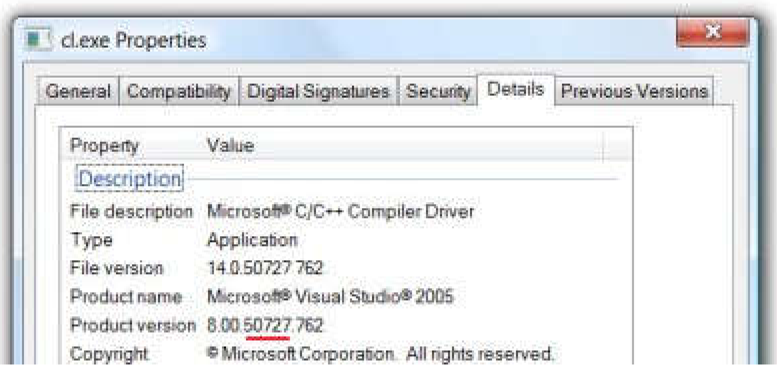
```
#define IMAGE_SIZEOF_SYMBOL                  18

//
// Section values.
//
// Symbols have a section number of the section in which they are
// defined. Otherwise, section numbers have the following meanings:
//

#define IMAGE_SYM_UNDEFINED           (SHORT)0          // Symbol is undefined or is
common.
#define IMAGE_SYM_ABSOLUTE            (SHORT)-1         // Symbol is an absolute value.
#define IMAGE_SYM_DEBUG               (SHORT)-2         // Symbol is a special debug
item.
#define IMAGE_SYM_SECTION_MAX         0xFEFF            // Values 0xFF00-0xFFFF are
special
```

The size of the structure is 18 bytes which is 0x12 in hex, the number noted earlier. Let's consider the "@comp.id" data layout given the structure above:

ShortName = @comp.id
Value = 0x00132359
SectionNumber = 0xFFFF (IMAGE_SYM_ABSOLUTE)
Type = 0x0000
StorageClass = 0x03
NumberOfAuxSymbols = 0x00

Exactly what dumpbin told us. From what I could understand the number of auxiliar symbols (NumberOfAuxSymbols) is the value which should be multiplied by 0x12 to get the next symbol in the table. So, if I have 1 auxiliar symbol, another 0x12 bytes belong to the current symbol before the next one comes. This is how you can enumerate the symbols in the table. This goes a little bit beyond the scope of this article, but now we know where the data comes from and how we can retrieve it from the library format.

On person made me notice that the low word of the comp.id value was the same as part of the version number of his VC++ compiler. Let's analyze for a second the fixed value inserted in the Rich Signature and let's consider its low word 0xC627 (50727). Can you spot it in the image?



Property	Value
Description	
File description	Microsoft® C/C++ Compiler Driver
Type	Application
File version	14.0.50727.762
Product name	Microsoft® Visual Studio® 2005
Product version	8.00.50727.762
Copyright	© Microsoft Corporation. All rights reserved.

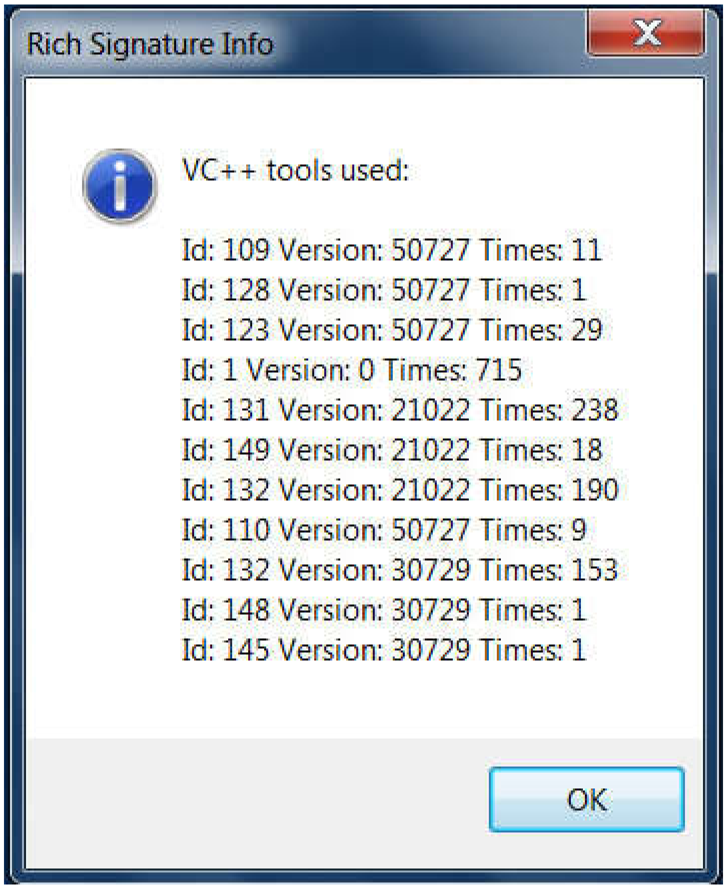
The high word contains the value of an internal enum and represents a product identifier. When I first wrote this paper I wasn't aware of this and it was later pointed out by someone who had access to the sources. So my guess about the meaning of the high word was incorrect. What follows is a short script to display the content of the Rich Signature.

- Download Rich Signature Displayer

```
-- Rich Signature Displayer
-- © 2008 Daniel Pistelli
```

```
filename = GetOpenFile()

if filename == null then
   return
end

hFile = OpenFile(filename)

if hFile == null then
   return
end

nSignDwords = 0

for i = 0, 100 do

   dw = ReadDword(hFile, 0x80 + (i * 4))

   if dw == null then
      return
   end

   -- is this the "Rich" terminator?

   if dw == 0x68636952 then

      nSignDwords = i

      break
   end
end

if nSignDwords == 0 then
   return
end

-- read xor mask

mask = ReadDword(hFile, 0x80 + ((nSignDwords + 1) * 4))

-- init string

strInfo = "VC++ tools used:\r\n"

-- decrypt versions

for i = 4, nSignDwords - 1, 2 do

   dw = ReadDword(hFile, 0x80 + (i * 4)) ^ mask

   id = dw >> 16
   minver = dw & 0xFFFF

   vnum = ReadDword(hFile, 0x80 + ((i + 1) * 4)) ^ mask

   strInfo = strInfo .. "\r\n" .. "Id: " .. id
       .. " Version: " .. minver .. " Times: " .. vnum
end


-- show versions

MsgBox(strInfo, "Rich Signature Info", MB_ICONINFORMATION)
```

Here's a screenshot with the output of this script:



```
VC++ tools used:

Id: 109 Version: 50727 Times: 11
Id: 128 Version: 50727 Times: 1
Id: 123 Version: 50727 Times: 29
Id: 1 Version: 0 Times: 715
Id: 131 Version: 21022 Times: 238
Id: 149 Version: 21022 Times: 18
Id: 132 Version: 21022 Times: 190
Id: 110 Version: 50727 Times: 9
Id: 132 Version: 30729 Times: 153
Id: 148 Version: 30729 Times: 1
Id: 145 Version: 30729 Times: 1
```

In the end, the value of @comp.id is quite harmless, compared to what people thought it could be.

Anyway, harmless or not, I'm now going to present you a neat little script which you may be pleased with just like I am. It does a lot of things. The main thing it does is to remove the DOS stub and the Rich Signature. It also strips the debug information (if any) present in the PE. As you might already know, current VC++ editions leave debug info even in release executable; contained in this debug info is the absolute path to the executable's pdb file, which gives away your project's current path. I hate that. It can be disabled from the compiler, but sometimes it comes handy to make sure. Finally, the script does all the rebuilding necessary to have a 100% working PE again, it even updates the checksum.

- Download Header Pack script

```
-- © 2008 Daniel Pistelli. All rights reserved.
-- Header Pack Script Version: 1.0.0.1

-- This neat little script does the following:
--
-- packs the dos header + PE header + section headers
-- removes useless things like the Rich Signature
-- removes linker references inside the PE header
-- strips the debug information (if any) from the PE
-- if it's a .NET, removes Strong Name Signature
-- updates checksum

-- shows an invalid PE Msgbox
function InvPEMsg()
   MsgBox("The current file seems to be an invalid PE. Cannot proceed!",
      strScriptName, MB_ICONEXCLAMATION)
end
```

```
-- ---------------------------------------------------
-- the main code starts here
-- ---------------------------------------------------

strScriptName = "Header Pack Script 1.0.0.1 - by Daniel Pistelli"

local filename = GetOpenFile("Select a PE...",
   "All\n*.*\nExe Files\n*.exe\nDll Files\n*.dll\n")

--[[ -- if it is a fixed file name, write: filename = @"C:\...\Release\App.exe" ]]

if filename == null then
   return
end

local hPE = OpenFile(filename)

if hPE == null then
   MsgBox("Couldn't open file.", "Error", MB_ICONEXCLAMATION)
   return
end

local OptHdrOffset = GetOffset(hPE, PE_OptionalHeader)

if OptHdrOffset == null then
   InvPEMsg()
   return
end

local bPE64 = IsPE64(hPE)
local bDotNET = IsDotNET(hPE)

-- ---------------------------------------------------
-- START PROCESSING
-- ---------------------------------------------------

-- PACK HEADERS

do
   local SecrHdrsOffset = GetOffset(hOriginalPE, PE_SectionHeaders)

   local SizeOfSections = GetNumberOfSections(hPE)
      * IMAGE_SIZEOF_SECTION_HEADER

   local FileHdrOffset = GetOffset(hPE, PE_FileHeader)

   local SizeOfOptionalHdr = ReadWord(hPE, FileHdrOffset + 16)

   local SizeOfPEHeader = 4 + 20 + SizeOfOptionalHdr;

   local PEHdrOffset = GetOffset(hPE, PE_NtHeaders)

   -- we assume that the PE header comes right after
   -- the Dos header, this is a normal PE

   FillBytes(hPE, 0x40, PEHdrOffset - 0x40, 0)

   -- move headers

   local HeadersSize = SizeOfPEHeader + SizeOfSections

   local PEHdrAndSects = ReadBytes(hPE, PEHdrOffset, HeadersSize)

   FillBytes(hPE, PEHdrOffset, HeadersSize, 0)
   WriteBytes(hPE, 0x40, PEHdrAndSects, HeadersSize)

   -- e_lfanew
   WriteDword(hPE, 0x3C, 0x40)

   OptHdrOffset = GetOffset(hPE, PE_OptionalHeader)

end

-- REMOVE LINKER REF

do

   WriteWord(hPE, OptHdrOffset + 2, 0x0000)

end

-- REMOVE DBG INFO

RemoveDebugDirectory(hPE)

-- REMOVE SNS IF .NET

if bDotNET == true then

   RemoveStrongNameSignature(hPE)

end

-- UPDATE CHECKSUM

UpdateChecksum(hPE)

-- SAVE FILE

SaveFile(hPE)
```

The header produced by this script comes, as I said, without DOS stub: I don't think it will be missing in 2008. The most efficient way to use this script is to execute it automatically after every linking. The PE header could be packed even more (for example one could reduce the data directory entries), but this goes beyond what I wanted to do: I just wanted my executables to be garbage clean.

Time to end this article. I enjoyed getting an insight into the libraries' file format. I might add support for it in the next version of the CFF Explorer, although I wouldn't consider this to be a priority, since most people wouldn't make use of it anyway.

Daniel Pistelli

** Apparently there had been already some information about the rich signature before I wrote this article. I did some google research before writing, but couldn't find anything about this topic. The prior information was signalled to me only after my paper became public. Here's a link to it. I mention it, because it's only fair, given my claim of the topic being undocumented.*