The Undocumented Microsoft "Rich" Header

Date: Mar 12, 2017
Last-Modified: Feb 28, 2018

SUMMARY:
There is a bizarre undocumented structure that exists only in Microsoft-produced executables. You may have never noticed the structure even if you've scanned past it a thousand times in a hex dump. This linker-generated structure is present in millions of EXE, DLL and driver modules across the globe built after the late 90's. This was when proprietary features were introduced into both Microsoft compilers and the Microsoft Linker to facilitate its generation. If you view the first 256 bytes of almost any module built with Microsoft development tools (such as Visual C++) or those that ship with the Windows operating system, such as KERNEL32.DLL from Windows XP SP3 (shown below), you can easily spot the signature in a hex viewer. Just look for the word "Rich" after the sequence "This program cannot be run in DOS mode":

```
00000000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00  MZ..............  <--DOS header
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 f0 00 00 00  ................
00000040  0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68  ........!..L.!Th  <--DOS STUB
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  17 86 20 aa 53 e7 4e f9 53 e7 4e f9 53 e7 4e f9  .. .S.N.S.N.S.N.  <--Start of "Rich" Header
00000090  53 e7 4f f9 d9 e6 4e f9 90 e8 13 f9 50 e7 4e f9  S.O...N.....P.N.
000000A0  90 e8 12 f9 52 e7 4e f9 90 e8 10 f9 52 e7 4e f9  ....R.N.....R.N.
000000B0  90 e8 41 f9 56 e7 4e f9 90 e8 11 f9 8e e7 4e f9  ..A.V.N.......N.
000000C0  90 e8 2e f9 57 e7 4e f9 90 e8 14 f9 52 e7 4e f9  ....W.N.....R.N.
000000D0  52 69 63 68 53 e7 4e f9 00 00 00 00 00 00 00 00  RichS.N.........  <--End of "Rich" header
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  50 45 00 00 4c 01 04 00 2c a1 02 48 00 00 00 00  PE..L...,..H....  <--PE header
```

When present, the "Rich" signature (DWORD value 0x68636952) can be found sandwiched (maybe "camouflaged" is a better word) between the DOS and PE headers of a Windows PE (portable executable) image. I say camouflaged, because it appears, perhaps by Microsoft's original design, to be part of the 16-bit DOS stub code, which it is not. Since many programmer probably weren't versed with 16-bit assembly even when Microsoft introduced this structure, you could argue the decision to embed something at this particular location in every executable was certainly a strategic one to help it hide in plain sight.

In Microsoft-linked executables, not only does the DOS mode string begin at predictable offset 0x4E, but the "Rich" structure always seems to appear at offset 0x80; this makes sense as the DOS header has probably been hardcoded for quite some time. Oddly enough, the "Rich" signature actually marks the end of the structure's data, whose size varies. Therefore the position of the signature as well as the total size of the structure changes from module to module. The 32-bit value that follows the signature not only marks the end of the structure itself, it happens to be the key that is used to decrypt the structure's data. Following this structure is the PE header, with a handful of zero-padded bytes in between.

Since this is an undocumented "feature" of the Microsoft linker, it is not surprising that there is no known option to disable it, short of patching (discussed below). At the time of discovery, all executables built using Microsoft language tools contained this structure (e.g. Visual C++, Visual Basic 6.x and below, MASM, etc.) causing many developers to fear the worst. Two "seemingly" identical installations of Visual Studio building the same source code appeared to produce executables with differing "Rich" headers. This combined with the fact that the structure was encrypted led many to the assumption that Microsoft was embedding personally identifiable information, ultimately allowing any given executable to be traced back to the machine it was built with. An old 2007 post on the Sysinternals forum refers to this structure the "Devil's Mark". Also of interest is a 2008 report on Donationcoder that Microsoft utilized the information from this structure as "evidence against several high-profile virus writers". A post from Garage4Hackers said "Microsoft uses compiler ids to prove that a virus is made on a particular machine with a particular compiler. Proving that the person owning the computer is the virus writer".

Note that while this structure is present in some .NET executables, it is not present in those that do not make use of the Microsoft linker. For example, an application composed purely of .NET Intermediate Language such as C# does not contain this structure. For any given executable module, you can check for the existence of the "Rich" header (in addition to viewing the decoded fields) using the pelook tool tool with the -rh option.

Before jumping to any conclusions, lets see what Microsoft is hiding here.

AN ARRAY OF NUMERIC VALUES:

First off, the "Rich" header really isn't a header at all. It is a self-contained chunk of data that doesn't reference anything else in the executable and nothing else in the executable references it. The structure was unofficially referred to as a header because it happens to reside in PE header area.

The structure happens to be little more than an array of 32-bit (DWORD) values between two markers. If one so chooses, the structure can even be safely zeroed-out from the executable without affecting any functionality. Just ensure you update the PE OptionalHeader's checksum if you alter any bytes in the file; although this is not necessary if the checksum field is zero (disabled).

Automated removal is possible through the peupdate tool. More recently, I found that Microsoft's editbin (in version 7.x and up) will also zero-out the "Rich" structure using the undocumented /nostub switch however this also removes the PE header offset from the DOS header effectively breaking the executable. Using editbin is therefore not recommended. Other removal options are discussed in the section, Patching the Microsoft Linker below.

In the KERNEL32.DLL sample above, the DWORD following the "Rich" sequence happens to have the value 0xF94EE753. This is the XOR key stored by and calculated by the linker. It is actually a checksum of the DOS header with the e_lfanew (PE header offset) zeroed out, and additionally includes the values of the unencrypted "Rich" array. Using a checksum with encryption will not only obfuscate the values, but it also serves as a rudimentary digital signature. If the checksum is calculated from scratch once the values have been decrypted, but doesn't match the stored key, it can be assumed the structure had been tampered with. For those that go the extra step to recalculate the checksum/key, this simple protection mechanism can be bypassed.

To decrypt the array, start with the DWORD just prior to the "Rich" sequence and XOR it with the key. Continue the loop backwards, 4 bytes at a time, until the sequence "DanS" (0x536E6144) is decrypted. This value marks the start of the structure, and in practice always seems to reside at offset 0x80. I think a lot of tools that parse the "Rich" structure rely on it starting at offset 0x80. I'd personally recommend against relying on this fact and parsing backwards from the "Rich" signature as described above to handle situations where this may not be the case. Since this is an undocumented structure, I think its best to avoid any assumptions such as hardcoded offsets, especially since you must search for the signature "Rich" anyway. With that said, I have yet to encounter an executable where offset 0x80 is not the start; that is, if the structure is present at all.

Following the decoding procedure using the KERNEL32.DLL sample shown above, we end up with following "Rich" structure where all values have been decrypted, and the array is listed beginning at offset 0x80 in ascending order:

```
OFFSET       DATA
------  ----------
  0080  0x536E6144  //"DanS" signature (decrypted) / START MARKER
  0084  0x00000000  //padding
  0088  0x00000000  //padding
  008C  0x00000000  //padding
  0090  0x00010000  //1st id/value pair     entry #1
```

```
0094  0x0000018A  //1st use count            id1=0,uses=394
0098  0x005D0FC3  //2nd id/value pair     entry #2
009C  0x00000003  //2nd use count            id93=4035,uses=3
00A0  0x005C0FC3  //3rd id/value pair     entry #3
00A4  0x00000001  //3rd use count            id92=4035,uses=1
00A8  0x005E0FC3  //4th id/value pair     entry #4
00AC  0x00000001  //4th use count            id94=4035,uses=1
00B0  0x000F0FC3  //5th id/value pair     entry #5
00B4  0x00000005  //5th use count            id15=4035,uses=5
00B8  0x005F0FC3  //6th id/value pair     entry #6
00BC  0x000000DD  //6th use count            id95=4035,uses=221
00C0  0x00600FC3  //7th id/value pair     entry #7
00C4  0x00000004  //7th use count            id96=4035,uses=4
00C8  0x005A0FC3  //8th id/value pair     entry #8
00CC  0x00000001  //8th use count            id90=4035,uses=1
00D0  0x68636952  //"Rich" signature      END MARKER
00D4  0xF94EE753  //XOR key
```

The array stores entries that are 8-bytes each, broken into 3 members. Each entry represents either a tool that was employed as part of building the executable or a statistic. You'll notice there are some zero-padded DWORDs adjacent to the "DanS" start marker. In practice, Microsoft seems to have wanted the entries to begin on a 16-byte (paragraph) boundary, so the 3 leading padding DWORDs can be safely skipped as not belonging to the data.

Each 8-byte entry consists of two 16-bit WORD values followed by a 32-bit DWORD. The HIGH order WORD is an id which indicates the entry type. The LOW order WORD contains the build number of the tool being represented (when applicable), or it may be set to zero. The next DWORD is a full 32-bit "use" or "occurrence" count.

THE ID VALUE:

The id value indicates the type of the list entry. For example, a specific id will represent OBJ files generated as a result of the use of a specific version of the C compiler. Different ids represent other tools that were also employed as part of building the final executable, such as the linker. Daniel Pistelli's article, Microsoft's Rich Signature (undocumented), found that the id values are a private enumeration that change between releases of Visual Studio. I have also found this to be the case, which unfortunately makes them a bit of a moving target to decipher.

Besides a couple exceptions which I'll explain below, the id is emitted by each compiler (or assembler) and is stored within each OBJ (and thus LIB) files linked against in the form of the "@comp.id" symbol. The "@comp.id" symbol happens to be short for "compiler build number" and "id". In fact, the DWORD value stored as the "@comp.id" symbol is the same DWORD being stored in the first half of applicable "Rich" list entries. I say applicable because not all list entries represent OBJ files.

Some ids can appear more than once in the list, while others do not. The id typically represents the following statistics:

- OBJ count for specific C compiler (cl.exe)
- OBJ count for specific C++ compiler (cl.exe)
- OBJ count for specific assembler (ml.exe)
- specific linker that built module (link.exe)
- specific resource compiler (rc.exe), when RES file linked
- imported functions count
- MSIL modules
- PGO Instrumented modules
- and so on...

Most of the entries above have an associated build number of the tool being represented, such as the compiler, assembler and linker. One exception to this is the imported functions count, which happens to be the total number of imported functions referenced in all DLLs. This is usually the only entry with a build number of zero. Note that the "Rich" structure does not store information on the number of static/private functions within each OBJ/source file.

The linker entry is always last in the list and represents the linker that built the module. The resource compiler, when present, is almost always 2nd to last in the list; next to the linker. Both the linker and resource compiler are represented by a hardcoded id and build values for each linker release. For example, when a resource script is employed, the linker uses the same id/build pair even if the RES is built from a resource compiler from another version of Visual Studio! Another oddity is the build value of the resource compiler entry typically does not match the build reported by the rc.exe command line. The correlation of unique build values to specific versions of Visual Studio tools is discussed in more detail below.

The linker seems to build most "Rich" structures in the following order, though not necessarily in the order appearing on the command line:

- Entries representing LIB files
- Entries representing individual OBJ files
- Resource Compiler
- Linker

At first glance you might guess that each referenced LIB file would represent one entry in the list, but this is not the case. The linker may may generate one or more entries for each LIB file depending on the number of unique "@comp.id" values found within. Since a LIB file is not much more than a concatenation of OBJ files, the resulting "count" member of these entries are the number of OBJ files referenced in the final executable that contain that exact "@comp.id" value. For example, statically linking to the Standard C Library usually generates assembler and C OBJ entries because that is what constitutes the source files internally used by Microsoft to build LIBCMT.LIB. When you link against this library, the unique "@comp.id" value-pairs are tallied together and the resulting counts are written to the list.

With that in mind, the "Rich" structure in KERNEL32.DLL can be annotated as follows:

id1=0,uses=394	id93=4035,uses=3	id92=4035,uses=1	id94=4035,uses=1	id15=4035,uses=5	id95=4035,uses=221	id96=4035,uses=4	id90=4035,uses=1
394 imports	???	???	1 rc script	5 asm sources	221 C sources	4 C++ sources	Linker

Here's another annotated example derived from a minimal C++ application linked with a resource script and the Standard C Library built from Visual C++ 7.1:

id15=6030,uses=20	id95=6030,uses=68	id93=2067,uses=2	id93=2179,uses=3	id1=0,uses=3;	id96=6030,uses=1	id94=3052,uses=1	id90=6030,uses=1
20 asm sources	68 C sources	???	???	3 imports	1 C++ source	1 rc script	Linker

Below is my attempt at a partial list of decoded ids from the version 6 and 7 Visual Studio toolsets based on a little trial and error. Note that many of the ids originate from the LIB files bundled with with the associated Visual C++ SDK versions, as the linker only hardcodes a few of the entries at the end of the list. It is also common to see a reference to MASM even when MASM is not utilized directly by a project as these references are pulled in automatically by the linker or SDK LIB files.

Microsoft Visual Studio 6.0 SP6

ID	MEANING
1	total count of imported DLL functions referenced; build number is always zero
4	seems to be associated when linking against Standard C Library DLL
19	seems to be associated when statically linking against Standard C Library
6	resource compiler; almost always last in list (when RES file used) and use-count always 1
9	count of OBJ files for Visual Basic 6.0 forms
13	count of OBJ files for Visual Basic 6.0 code
10	count of C OBJ files from specific cl.exe compiler
11	count of C++ OBJ files from specific cl.exe compiler
14	count of assembler OBJ files originating from MASM 6.13
18	count of assembler OBJ files originating from MASM 6.14
42	count of assembler OBJ files originating from MASM 6.15

Microsoft Visual Studio 7.1 SP1

ID	MEANING
1	total count of imported DLL functions referenced; build number is always zero; same as in Linker versions 5.0 SP3 and 6.x
15	count of assembler OBJ files originating from MASM 7.x
90	linker; always present and always at end of list; use-count always 1
93	Always seems to be present no matter how the executable was built, but doesn't appear to originate from @comp.id symbols (???)
94	resource compiler; almost always 2nd to last in list (when RES file used) and use-count always 1
95	count of C OBJ files from specific cl.exe compiler
96	count of C++ OBJ files from specific cl.exe compiler

Not only do the ids change with each major linker release (sometimes with service packs too), but newer versions of the SDK's LIB files use different and higher id numbers for the same thing, such as the C and C++ compiler. So not only do the build numbers change with each SDK, but the id identifying the type of entry also changes. Unless the idea is to make the header difficult to interpret, the id may be meant to be combined with the build number to provide a unique compiler-that-built-SDK instance statistic which could be used to trace leaked or BETA versions of tools or SDKs. The whole system might also double as another check system, where if the tool ids and reported build versions don't match known publicly released pairs, this would be another indication the entries in the list were tampered with. Without any official word from Microsoft, some of this is pure speculation.

The good news is that if you are only interested in detecting modern versions of Visual Studio (7.x and up), the id member can be completely ignored! More information about detection is presented below.

WHEN DID MICROSOFT INTRODUCE THE "RICH"-ENABLED LINKER?

The short answer is in 1998, with Visual Studio 6.0 (LINK 6.x). The long answer is the final Service Pack for Visual Studio 5.0; that is, the version 5.10.7303 linker introduced with SP3 in 1997 was the first "Rich" capable linker. The catch was that the list this linker produced was practically empty because the compilers at the time (e.g. Visual C++ 5.x, MASM 6.12) did not yet emit the "@comp.id" symbol to the OBJ files. Not surprisingly, the LIB files that shipped with the product's SDK were also missing the "@comp.id" symbol. The result was a "Rich" structure with either a single entry for the imports, or with an additional entry to represent a compiled resource script.

If you however link a Visual C++ 6.0 OBJ file with the older 5.0 SP3 (5.10.7303) linker, you will get a proper "Rich" structure because the 6.0 OBJ file contained the "@comp.id" symbol with build information. The 6.0 OBJ files were however incompatible the 5.0 SP2 and earlier linkers; if you attempted to link-in any of these modules using the older linkers, you would run in to error: LNK1106: "invalid file or disk full: cannot seek to 0xXXXXXXXX". This is an indication that in 1997, Microsoft changed the OBJ file format.

In summary, the Visual C++ 5.0 SP3 linker and the linker that would be released next with Visual C++ 6.0, both supported a new type of OBJ file. Specifically, the OBJ files that would facilitate the generation of the this new "Rich" structure.

CHANGELIST:

PRODUCT VERSION	YEAR	CHANGE
Visual Studio 97 (5.0) SP3	1997	First linker capable of producing "Rich" header and supporting new OBJ format to be released with the not-yet-public VC++ 6.0 compiler (cl.exe 12.x); however compiler's at the time did not yet support writing "@comp.id" to OBJ files so the list had minimal information
Visual Studio 6.x	1998	Microsoft compilers, including Visual Basic now support writing "@comp.id" symbol to OBJ files; bundled SDK LIB files now contain "@comp.id" build information; as a result, executables built using the Visual C++ 6.0 compiler and linker now get the first "proper" "Rich" headers.
Visual Studio 7.0 .NET (2002)	2002	Linker now appends its own entry to the list and is always last; fortunately we now have a predictable entry that is retained in future versions

BUILD NUMBERS FOR DETECTION:

Before continuing further, I want to stress an important point. There is little preventing someone from either tampering with or completely falsifying the "Rich" header. While this structure may provide useful information for the majority of executables, other signature methods should be utilized in conjunction where accuracy is paramount. Some of these methods may include searching for specific patterns in the headers and/or analysis of the entry-point code. For example the Borland and Watcom linkers can be identified by specific patterns unique to each from their DOS stubs. The presence of a "Rich" header, or lack thereof, doesn't mean Microsoft's linker cannot be detected by other clues.

A typical version number for a Microsoft product consists of major and minor numbers (one byte each) followed by a 16-bit build number and sometimes another 16-bit sub-version number. Since we primarily have the build number for each "Rich" entry to go by, how might we distinguish a specific version of Visual Studio from this information? There are at least 3 ways:

- The MajorLinkerVersion and MinorLinkerVersion members of the PE's OptionalHeader can be combined with the last entry in the "Rich" list (if MajorLinkerVerion >= 7) to construct the full version of the linker. Once the linker is known, one can assume the version of Visual Studio including *that* linker was responsible for building the executable even if not all inputs came from this version.
- The build numbers for each release of Visual Studio are almost completely unique which allow build numbers to identify a specific version of the toolset used; that is for id's besides the linker. The one exception I'm aware of is build number 50727. This build number was issued to public releases of both Visual Studio 2005 and 2012. As mentioned above, you might make the distinction by checking the PE MajorLinkerVersion and testing it for 8 and 11 respectively to at least determine the full version of the linker.
- Because the entry type ids change between releases of Visual Studio, the id in combination with the build number can be used to uniquely identify a version of Visual Studio.

Based on the information above, if you want to detect versions of Visual Studio 7.0 and up, things couldn't be easier. If the MajorLinkerVersion in the PE's OptionalHeader is 7 or greater, indicating Visual Studio .NET 7.0 (2002) and up, the last entry in the list always represents the linker that built the module. If that build number corresponds to a known version of Visual Studio, you might consider it safe to assume the compiler is also from the same toolset.

As for versions of Visual Studio supporting the "Rich" structure prior to 7.0, the detection rules were a little different because no linker entry was written to the end of the "Rich" list. Perhaps Microsoft figured it was enough that the PE header contained the Major and Minor version for the linker and the build number was not important enough to include.

It is interesting to note that the id and build versions embedded within the publicly shipped SDK LIB files are those of non-public releases of Microsoft compilers; this makes sense because Microsoft builds its SDKs internally, but this happens to be a good thing for detection. This allows us to distinguish the SDK's compiler id/build pairs from those pairs that represent the compilers responsible for building the executable. In other words, if you know where the linker entry is and the entries that represent the SDK LIB files because they are not recognized public versions (see table below), the only thing left are the compiler entries we want to use for detection! You will then be able to determine the language used to build an executable, whether it be C, C++, MASM or all in combination in addition to the toolset version of each. All of this is assuming you want to use the build numbers alone for detection rather than hardcoding the differing tool ids per version of Visual Studio to combine with the build numbers.

Going back to the KERNEL32.DLL example above, we can see the last entry's build number is 4035 which corresponds to one of the known public Microsoft 7.1 linkers. Using a lookup table, such as that shown below, applications can use this information to correlate [mostly] unique build numbers to known Microsoft Visual Studio toolsets.

MASM 6.x BUILDS

BUILD	PRODUCT VERSION
7299	6.13.7299
8444	6.14.8444
8803	6.15.8803
9030	6.15.9030 (VS.NET 7.0 BETA 1)

Visual Basic 6.0 BUILDS

BUILD	PRODUCT VERSION
8169	6.0 (also reported with SP1 and SP2)
8495	6.0 SP3
8877	6.0 SP4
8964	6.0 SP5
9782	6.0 SP6 (same as reported by VC++ but different id)

VISUAL STUDIO BUILDS

BUILD	PRODUCT VERSION	CL VERSION	LINK VERSION
8168	6.0 (RTM, SP1 or SP2)	12.00.8168	6.00.8168
8447	6.0 SP3	12.00.8168	6.00.8447
8799	6.0 SP4	12.00.8804	6.00.8447
8966	6.0 SP5	12.00.8804	6.00.8447
9044	6.0 SP5 Processor Pack	12.00.8804	6.00.8447
9782	6.0 SP6	12.00.8804	6.00.8447
9030	7.0 2000 (BETA 1)	13.00.9030	7.00.9030
9254	7.0 2001 (BETA 2)	13.00.9254	7.00.9254
9466	7.0 2002	13.00.9466	7.00.9466
9955	7.0 2002 SP1	13.00.9466	7.00.9955
3077	7.1 2003	13.10.3077	7.10.3077
3052	7.1 2003 Free Toolkit	13.10.3052	7.10.3052
4035	7.1 2003	13.10.4035 (SDK/DDK?)	
6030	7.1 2003 SP1	13.10.6030	7.10.6030
50327	8.0 2005 (Beta)	?	?
50727 (linkver 8.x)	8.0 2005	14.00.50727.42 14.00.50727.762 SP1?	
21022	9.0 2008	15.00.21022	
30729	9.0 2008 SP1	15.00.30729.01	
30319	10.0 2010	16.00.30319	
40219	10.0 2010 SP1	16.00.40219	
50727 (linkver 11.x)	11.0 2012	17.00.50727	
51025	11.0 2012	17.00.51025	
51106	11.0 2012 update 1	17.00.51106	
60315	11.0 2012 update 2	17.00.60315	
60610	11.0 2012 update 3	17.00.60610	
61030	11.0 2012 update 4	17.00.61030	
21005	12.0 2013	18.00.21005	
30501	12.0 2013 update 2	18.00.30501	
31101	12.0 2013 update 4	18.00.31101.0	
40629	12.0 2013 SP5	18.00.40629 SP5	
22215	14.0 2015	19.00.22215 Preview	
23026	14.0 2015	19.00.23026.0	
23506	14.0 2015 SP1	19.00.23506 SP1	
23824	14.0 2015 update 2	(unverified)	
24215	14.0 2015	19.00.24215.1 (unverified)	
24218	14.0 2015	19.00.24218.2	
25019	14.1 2017	19.10.25019.0	

NOTE: The table above was compiled from various sources; it is not an exhaustive list.

BUILD NUMBERS DON'T ALWAYS MATCH REPORTED COMMAND LINE/ VERSION RESOURCE VALUES!

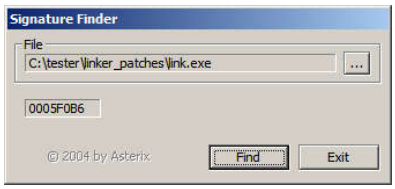
As you can see, the build numbers don't always correspond to what is reported from the command line. For example cl.exe for Visual C++ 6.0 reports version 12.00.8804 for Service Packs 4 thru 6, however the "@comp.id" value written to OBJ files is different for each service pack, such as 8799,8966,9044, and 9782 for SP4, SP5, SP5 (Processor Pack) and SP6 respectively. You can see the same pattern in Visual C++ 7.x. This allows for unique detection for each Service Pack.

PATCHING THE MICROSOFT LINKER:

Rather than using a tool (such as peupdate) to remove the "Rich" header on a per-executable basis, it is possible to "fix" the linker so that the "Rich" header is never written in the first place.

It wasn't long between the "Rich" header's discovery gone public and the appearance of a linker patch to prevent the structure from being written to the executable. This is a cleaner solution than manually zeroing-out each executable produced, however a new patch is needed for each version of the linker. As an added bonus, patching reclaims the area originally occupied by the "Rich" header (usually offset 0x80) as the spot where the PE header will instead be placed. This can reduce the size of the executable depending on the file alignment value passed to the linker.

In August of 2005, there was a PE tutorial written by Goppit that briefly describes using a tool called Signature Finder to patch the Linker. This is a simple GUI tool that when supplied the path to LINK.EXE, locates the RVA address of the CALL instruction for the routine which generates the "Rich" Header.



Knowing where the "Rich" routine is invoked by the linker is the first step; how to patch is up to you. However the traditional patch method is to NOP-out the ADD instruction following the CALL.

To do this, load LINK.EXE in a disassembler or debugger and navigate to the location reported by the tool (adding a 0x400000 base address to the reported RVA). If you have symbols loaded, you'll see disassembly similar to the following within the IMAGE::BuildImage() function:

```
0045F0A5  E8 56 45 FC FF        call    ?UpdateCORPcons@@YGXXZ                  ; UpdateCORPcons(void)
0045F0AA  55                    push    ebp                                     ; struct IMAGE *
0045F0AB  E8 30 D2 01 00        call    ?UpdateSXdata@@YGXPAVIMAGE@@@Z          ; UpdateSXdata(IMAGE *)
0045F0B0  8D 54 24 14           lea     edx, [esp+448h+lpMem]
0045F0B4  52                    push    edx
0045F0B5  55                    push    ebp
0045F0B6  E8 45 A6 FF FF        call    ?CbBuildProdidBlock@IMAGE@@AAEKPAPAX@Z  ; IMAGE::CbBuildProdidBlock(void**) <--- BUILDS the "Rich" Header
0045F0BB  8B 8D 3C 02 00 00     mov     ecx, [ebp+23Ch]
0045F0C1  03 C8                 add     ecx, eax                                ; <--- NOP this out!
0045F0C3  89 44 24 2C           mov     [esp+448h+var_41C], eax
0045F0C7  89 8D 40 02 00 00     mov     [ebp+240h], ecx
0045F0CD  FF 15 BC 12 40 00     call    ds:__imp___tzset
```

The "Rich" Header routine identified by the tool is named CbBuildProdidBlock(); we can now assume Microsoft internally refers to the "Rich" structure as the "Product ID Block". If the ADD instruction below it (address 0x45F0C1) is changed from bytes "03 C8" to "90 90" (NOPs), the linker still internally generates the structure, but because we've removed the instruction that advances the current file position, the PE header (which comes next in the image) overwrites the "Rich" structure. Problem solved, no information leak.

If you don't want to run the Signature Finder tool, below is a table with patch address information for all of the publicly-released 6.xx and 7.xx Microsoft linkers. The location is for the "ADD ECX,EAX" instruction (bytes "03 C8"). To perform the patch, replace the ADD instruction with two NOP bytes ("90 90"). This can be done with the bytepatch tool using the following command line:

```
bytepatch -pa <address> link.exe 90 90      //using address column below
```

OR

```
bytepatch -a <offset> link.exe 90 90        //using offset column below
```

Replace <address> or <offset> values with those corresponding in the ADDRESS and OFFSET columns below for the linker you are using:

VERSION	SHIPPED WITH	MD5	FILE SIZE	ADDRESS	OFFSET
LINK.EXE 6.00.8168	MSVC 6.0 RTM,SP1,SP2	7b3d59dc25226ad2183b5fb3a0249540	462901	0x44551A	0x4551A
LINK.EXE 6.00.8447	MSVC 6.0 SP3,SP4,SP5,SP6	24323f3eb0d1afa112ee63b100288547	462901	0x445826	0x45826
LINK.EXE 7.00.9466	MSVC .NET 7.0 (2002) RTM	dbb5bf0ce85516c96a5cbdcc3d42a97e	643072	0x45CD82	0x5CD82
LINK.EXE 7.00.9955	MSVC .NET 7.0 (2002) SP1	2042a0f45768bc359a5c912d67ad0031	643072	0x45CD32	0x5CD32
LINK.EXE 7.10.3052	MSVC .NET Free Toolkit	8d7a69e96e4cc9c67a4a3bca1b678385	647168	0x45EA0F	0x5EA0F
LINK.EXE 7.10.3077	MSVC .NET 7.1 (2003)	4677d4806cd3566c24615dd4334a2d4e	647168	0x45EA0F	0x5EA0F
LINK.EXE 7.10.6030	MSVC .NET 7.1 (2003) SP1	59572e90b9fe958e51ed59a589f1e275	647168	0x45F0C1	0x5F0C1
LINK.EXE 10.00.30319.01	MSVC .NET 10 (2010) RTM	d358960cb06c16476e89bd808a5e67fa	852296	0x476DBD	0x761BD
LINK.EXE 10.00.40219.01	MSVC .NET 10 (2010) SP1	60f3d6c30c1bbfb4da62df454a4be470	851272	0x476CAD	0x760AD

Unfortunately, the Signature Finder tool only works with Microsoft linkers prior to and including Visual Studio .NET 7.1 (2003). RE Analysis of the tool indicates that it searches up to 4 possible linker signatures (all known linkers available at the time of the tool's release in 2004), so trying to patch a newer linker such as the one that shipped with MSVC .NET 8.0 (2005), results in an error. However, manually finding the location using a disassembler is not difficult. I received an e-mail from icestudent with a method he uses to manually patch each Microsoft linker release from 8.0 and up. Here is a break-down of this method:

- Ensure your symbol path is set correctly; then download symbols for the linker you want to patch; e.g.: symchk /v LINK.EXE
- open LINK.EXE with IDA Pro
- Open the imports window, locate "_tzset", and go to it
- Open the references for "_tzset" (CTRL-X) and go to the "IMAGE::BuildImage" reference (or the "IMAGE::GenerateWinMDFile" for CLR executables).
- Around the "CALL _tzset" instruction, locate the "CALL IMAGE::CbBuildProdidBlock" instruction. In older versions of the linker it was closer and above "_tzset", in modern versions it is below and quite far.
- If you don't have symbols, check for all CALL instructions around "_tzset" and find the one where the referenced function begins with a call to "HeapAlloc"; this will be the "IMAGE::CbBuildProdidBlock()" function.
- After the "CALL IMAGE::CbBuildProdidBlock", you will see some code like "MOV reg, ...", "ADD reg, reg2", "MOV [mem], reg". NOP-out the second ADD instruction (or sometimes LEA) which is responsible for adjusting the PE offset in memory past the Rich signature.

If you don't use the method above, the table below contains the patch offsets for some post MSVC 7.x linkers. Thanks goes to icestudent for this information!

32-BIT LINK.EXE

x86 VERSION	OFFSET	ORIGINAL BYTES	PATCH BYTES
8 SP1	0x6A382	03 D0	90 90
9 RTM	0x6A20F	03 C8	90 90
9 SP1	0x6BE7F	03 C8	90 90

64-BIT LINK.EXE

x64 VERSION	OFFSET	ORIGINAL BYTES	PATCH BYTES
8 RTM	0x8157A	93	8B
8 SP1	0x80DAC	93	8B
9 RTM	0x78205	93	8B

10 B1	0x6CF50	03 C8	90 90
10 B2	0x75EED	03 D0	90 90
10 CTP	0x6C26D	03 C8	90 90
10 RTM	0x761BD	03 D0	90 90
10 SP1	0x760AD	03 D0	90 90
11 RTM U1	0x235BF	03 CE	90 90
11 RTM	0x17AEF	03 CE	90 90
vc18 CTP2	0x31920	03 CB	90 90
vc18 PREVIEW	0x1B3D8	03 CF	90 90
vc18 RC1	0x27B43	03 CF	90 90
vc18 RTM	0x31168	03 CB	90 90

9 SP1 KB	0x78CE1	8D 14 0E	90 90 90
9 SP1	0x78CE5	93	8B
10 B1	0x7A01F	93	8B
10 B2	0x7A09D	93	8B
10 SP1	0x7A06D	93	8B
11 RTM U1	0x136B5	03 D0	90 90
11 RTM	0x136B5	03 D0	90 90
vc18 CTP2	0xE22A	03 D0	90 90
vc18 PREVIEW	0x853D	03 D0	90 90
vc18 RC1	0xE684	03 D0	90 90
vc18 RTM	0xEF3A	03 D0	90 90

To patch using the offsets in the table above, use the following bytepatch command line, replacing <file-offset> and <patch-bytes> with the appropriate entry:

```
bytepatch -a <file-offset> link.exe <patch-bytes>
```

CONSPIRACY THEORIES:

When the public first became aware of the "Rich" header, the obvious encryption of this structure of unknown information made a lot of people nervous and suspicious. Because Microsoft never officially confirmed the existence of this structure, their lack of transparency made a lot of developers assume the worst. Here you can have an identical-source program built on two different machines and end up with a slightly different executable because the information contained within the "Rich" header was different. It is not surprising that people assumed Microsoft was embedding machine or otherwise personally identifiable information within the structure. These might include a NIC/MAC address, a CPU identifier, Windows registration information or even a unique GUID representing a particular installed instance of a Microsoft product or operating system.

In reality, the only thing stored here are the build numbers for the Microsoft-specific tools responsible for a specific component in an executable module. The slightest difference in Visual Studio version, SDK version or 3rd party libraries used will cause an alteration of the "Rich" header.

The PE/COFF specification defines a minimum file alignment of 512 bytes. Since this value leaves more than enough room for an executable's header section to fully contain the DOS and PE headers, there will always be leftover wasted space between the headers section and the subsequent section. Microsoft capitalized on this fact by inserting the "Rich" header in the padding space, since it wouldn't generally affect the final executable size one way or the other.

To Microsoft's credit, the "Rich" header offers invaluable debugging statistics about how a given executable was built. Because the Visual C/C++ compiler and linker command lines are probably among the most complex command lines of any of Microsoft products to date, not to mention the different versions of those tools available and combinations of SDKs that can be used, a structure such as the "Rich" header being embedded within every executable could certainly save countless man hours in debugging complex build environment problems. Did I mention Microsoft's internal build environment is among the most complex in the world?

If Microsoft's case against the author of a virus hinged on the virus being created by a particular version of Visual Studio that matched the version on a confiscated machine, I guess the "Rich" header could be used as evidence to prove this fact but probably not much more. There are other useful reasons Microsoft might want to bury such a secret "fingerprint" within executables. If Microsoft could prove which versions of certain libraries were employed, this would help them to assert intellectual property rights or even a redistribution license violation as they could distinguish between public, beta and pre-release versions. If companies used beta versions of Microsoft tools or libraries to release executables to the public outside of a specific time period, Microsoft would now have a way to find out. The "Rich" header could also help ensure publicly released benchmark tests were done fairly on properly built, Microsoft-sanctioned executables. These reasons could have been a bigger deal at the time the "Rich" header was invented than they are today.

The problem was that Microsoft intentionally hid and encrypted this information. Since the structure doesn't officially exist, there isn't going to be an official way to disable it. Anyone who develops with Microsoft tools gets this structure crammed in their executable whether or they like it or not. Failing to document this fact can be considered a questionable practice.

However, once people realized Microsoft wasn't embedding personally identifiable information in their executables, the "Rich" header was no longer the hot topic it once was.

ORIGINS OF "RICH" AND "DANS" SEQUENCES:

According to a 2012 post on Daniel Pistelli's RCE Cafe blog, information from two people who claimed to have worked on the Microsoft Visual C++ team said the word "Rich" likely originated from "Richard Shupak", a Microsoft employee who worked in the research department and had a hand in the Visual C++ linker/library code base. NOTE: Richard Shupak is listed as the author at the top of the file PSAPI.H in the Platform SDK. The PSAPI library (The NT "Process Status Helper" APIs) retrieves information about processes, modules and drivers.

"DanS" was likely attributed to employee "Dan Spalding" who presumably ran the linker team. I can vouch for the fact that there was a "Dan Spalding" employee working on the Visual C++ team around the turn of the century. Apparently their initials also show up in the MSF/PDB format!

CONCLUSION AND REFERENCES:

The first known public information about this structure goes back to at least July 7th, 2004 from the article, Things They Didn't Tell You About MS LINK and the PE Header, a loose specification authored by "lifewire". I've archived the article here, because it is no longer available at one of the original links. While the article was brief, it was densely packed with useful details, such as the layout of the "Rich" structure and how the checksum key is calculated. It is not mentioned how the author came to know such information, but information like this is usually leaked or derived from reverse engineering. At the end of the article, he attributes the "Dan^" sequence as being a reference to Microsoft employee "Dan Ruder", but the sequence was actually and has always been "DanS", so I think this conclusion is incorrect.

When I was writing the pelook tool and was looking to add minimal compiler signature detection, I initially stumbled upon Daniel Pistelli's excellent 2008 article, titled Microsoft's Rich Signature (undocumented). This article describes what he discovered while reverse engineering Microsoft's linker. Pistelli's research was independent of lifewire's 2004 article which was unknown to him at the time. Despite this, he arrived at the same conclusion.

Pistelli's article was the first I'd heard of such a structure. I was surprised to learn of its existence and that it had been right under my nose all of those years. I was even more surprised that further information (official or unofficial) was not available. My goal in writing this article was to fill in some of the gaps of information not previously available, such as how far back Microsoft's linker had support for the "Rich" structure and how it changed between different versions of Visual Studio.

Other links I found useful:

- A tutorial from 2010 that was based off of the original "lifewire" article.
- A posting on asmcommunity
- A posting on trendystephen

<END OF ARTICLE>

Questions or Comments?

Changelist for this document

```
2018-08-27     * added VS.NET 7.0 BETA 1 and BETA 2 entries to MASM 6.x and Visual Studio tables
               * added several missing 14.x and 12.0 builds to Visual Studio table

2018-08-26     * added patch info for link.exe 10.00.30319.01 (MSVC 2010 RTM); courtesy Vernon Brooks
               * added patch info for link.exe 10.00.40219.01 (MSVC 2010 SP1)
               * added patch cmdline variation for offset column

2018-02-28     * typos fixed and minor rewording for clarification

2017-11-22     * added information about editbin 7.x (and up) undocumented /nostub switch to remove the "Rich" header

2017-11-12     * added links to peupdate tool (for "Rich" header removal)

2017-06-22     * added section "Patching the Microsoft Linker"; added 8.0 2005 Beta to MSVC BUILDS

2017-04-10     * added NOTE that Richard Shupak is listed as the author at the top of PSAPI.H (from the Platform SDK)

2017-03-28     * added table for VB6 build numbers and filled in their ids for the Visual Studio 6.0 table
               * changed VS6/VS7 id table references from "linker" to "Visual Studio toolset" as the linker is not
                 responsible for all of the id values; added that it is common to see MASM references when MASM not
                 used directly in a project.
```

1:1