RICH HEADERS: LEVERAGING THIS MYSTERIOUS ARTIFACT OF THE PE FORMAT

Michal Poslušný & Peter Kálnai
ESET, Czech Republic

{michal.poslusny, peter.kalnai}@eset.cz

ABSTRACT

Ever since the release of *Visual Studio 97 SP3*, *Microsoft* has placed an undocumented chunk of data between the DOS and PE headers of every native Portable Executable (PE) binary produced by its linker without any possibility to opt out. The data contains information about the build environment and the scale of the project, stored in a simple yet effective way using blocks of the following values: a product identifier, its build number, and the number of times it was used during the build process. Several research papers on this topic have been released over the years, coming up with the name 'Rich Header' and shedding some light on its purpose and structure, but we feel that it has never been used to its full potential by the security industry.

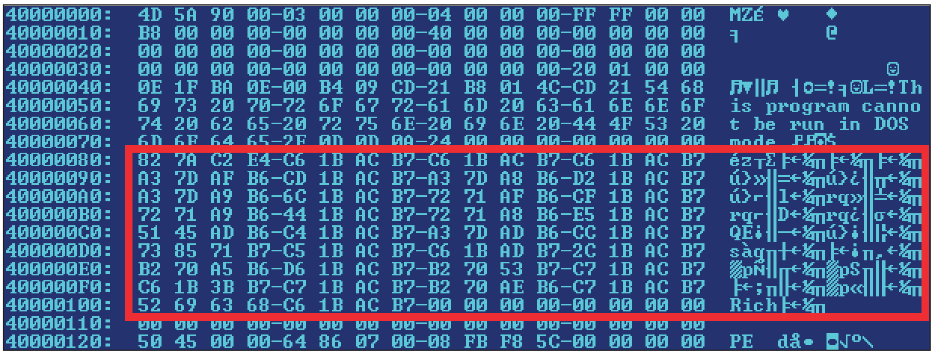
When an analyst encounters a rare custom malware sample involved in, say, an APT, and is clutching at straws to draw conclusions about the case, this mysterious structure could provide some reasonable clues. Not only does it reveal the type of components involved in the project behind the malware and the build tools used, but, forming an abounding set of variations, it also helps with locating similar samples. We introduce a hierarchy of similarity levels, together with real-world examples where they have successfully been applied.

For various crimeware kits, which are (re)distributed on a daily basis, the header could indicate whether their source code is available more widely or under the control of a single actor. Moreover, the headers from their encapsulating malware packers often manifest their own anomalies and could cluster a larger set of samples of the same nature. These inconsistencies could easily be identified and turned into heuristics based on the situation, such as: an unusual offset or the size of the header, an invalid identifier or its combination with the build version, the image size not corresponding to the magnitude of the project, etc.

We will also showcase our in-house designed database infrastructure and the tooling we've built around it: similarity lookup, a rule-based notification system for malware hunting and the detection of anomalies. The database currently holds the Rich Header information for tens of millions of executables and continuously processes a live feed of new, incoming files. It is currently growing by ~180,000 unique records per day.

INTRODUCTION

PE 'Rich Headers' were introduced with the release of *Visual Studio 97 SP3*. *Microsoft* didn't announce that it had implemented such a feature or give a reason for its introduction, and never released any kind of documentation for it, so we cannot really be sure about its original purpose, but it seems that *Microsoft* simply wanted to have some sort of development environment fingerprint stored in the executables, or perhaps to help with diagnostics and debugging. Regardless of the original intent, the Rich Header has proved to be a very valuable block of data for malware researchers, where a few hundred bytes, when interpreted correctly, can be used as a very strong factor for attribution and detection.



```
40000000:  4D 5A 90 00-03 00 00 00-04 00 00 00-FF FF 00 00  MZÉ ♥   ◆
40000010:  B8 00 00 00-00 00 00 00-40 00 00 00-00 00 00 00  ╕       @
40000020:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
40000030:  00 00 00 00-00 00 00 00-00 00 00 00-20 01 00 00              ☺
40000040:  0E 1F BA 0E-00 B4 09 CD-21 B8 01 4C-CD 21 54 68  ♫▼║♫ ┤○═!╕☺L═!Th
40000050:  69 73 20 70-72 6F 67 72-61 6D 20 63-61 6E 6E 6F  is program canno
40000060:  74 20 62 65-20 72 75 6E-20 69 6E 20-44 4F 53 20  t be run in DOS
40000070:  6D 6F 64 65-2E 0D 0D 0A-24 00 00 00-00 00 00 00  mode.♪♪◙$
40000080:  82 7A C2 E4-C6 1B AC B7-C6 1B AC B7-C6 1B AC B7  éz┬Σ╞←¼╖╞←¼╖╞←¼╖
40000090:  A3 7D AF B6-CD 1B AC B7-A3 7D A8 B6-D2 1B AC B7  ú}»╢═←¼╖ú}¿╢╥←¼╖
400000A0:  A3 7D A9 B6-6C 1B AC B7-72 71 AF B6-CF 1B AC B7  ú}⌐╢l←¼╖rq»╢╧←¼╖
400000B0:  72 71 A9 B6-44 1B AC B7-72 71 A8 B6-E5 1B AC B7  rq⌐╢D←¼╖rq¿╢σ←¼╖
400000C0:  51 45 AD B6-C4 1B AC B7-A3 7D AD B6-CC 1B AC B7  QE¡╢─←¼╖ú}¡╢╠←¼╖
400000D0:  73 85 71 B7-C5 1B AC B7-C6 1B AD B7-2C 1B AC B7  sàq╖┼←¼╖╞←¡╖,←¼╖
400000E0:  B2 70 A5 B6-D6 1B AC B7-B2 70 53 B7-C7 1B AC B7  ▓pÑ╢╓←¼╖▓pS╖╟←¼╖
400000F0:  C6 1B 3B B7-C7 1B AC B7-B2 70 AE B6-C7 1B AC B7  ╞←;╖╟←¼╖▓p«╢╟←¼╖
40000100:  52 69 63 68-C6 1B AC B7-00 00 00 00-00 00 00 00  Rich╞←¼╖
40000110:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
40000120:  50 45 00 00-64 86 07 00-08 FB F8 5C-00 00 00 00  PE  då• ◘√°\
```

Figure 1: PE binary with Rich Headers highlighted.

The structure

The meaning of the 'header' is actually quite straightforward. In Figure 2 we can see the structure of the header, as described by *Microsoft* in the *Windows 2000* source code leak [1]. The data is inserted after the `IMAGE_DOS_HEADER` and its stub and before the beginning of the PE header (`IMAGE_NT_HEADERS` structure).

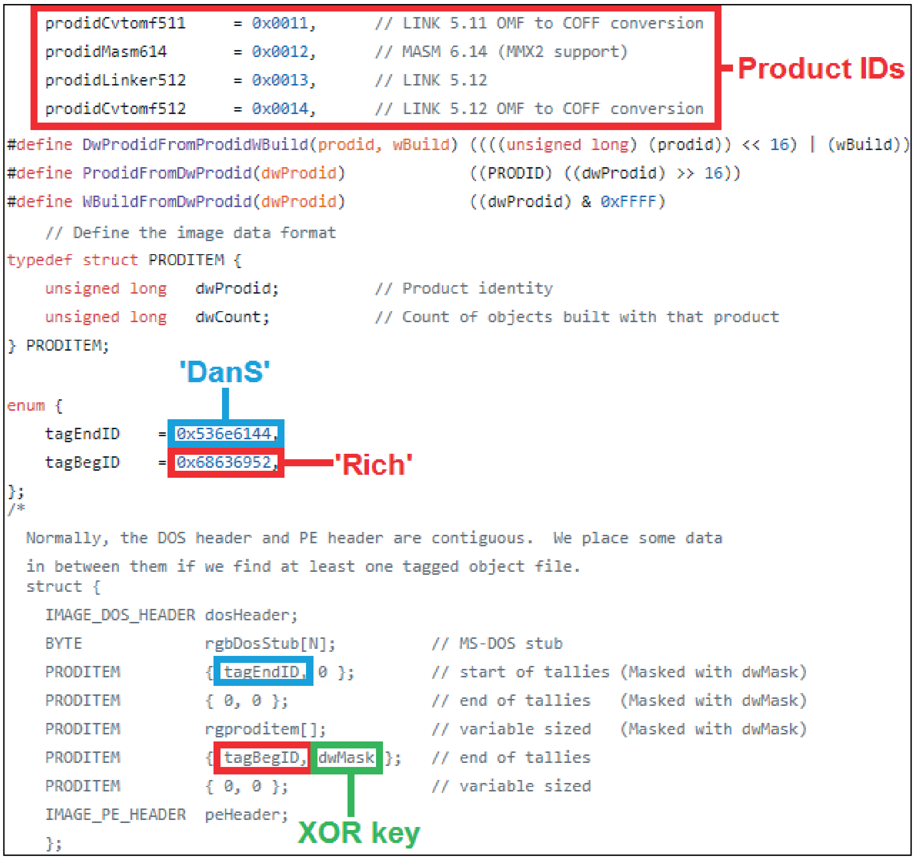
Its end is marked by the `Rich` keyword (0x68636952) followed by an XOR key DWORD, which was used to encrypt the rest of the header. The beginning of the plaintext header is marked by the `DanS` keyword (0x536E6144), followed by padding of three null DWORDs.

As we can see in Figure 2, the data is an array of simple 64-bit `PRODITEM` structures that consist of two 32-bit fields: `dwProdid` and `dwCount`. The `dwProdid` field is a combination of a product identifier (`ProdID`) and its corresponding `Build` number and can be considered as two separate 16-bit values.

`ProdID` represents a product from the *Visual Studio* development toolchain – these include objects like linkers, C/C++ compilers, MASM, resources, imports, implibs, etc. One is assigned to every object file upon creation and eventually all `ProdID`s from every object file used in a project are collected and written into the resulting binary, by the linker, during the linking process.

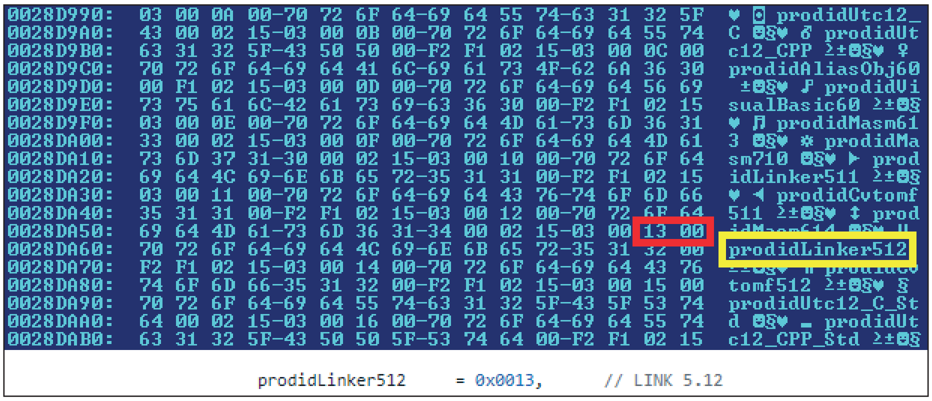
Since `ProdID`s are an enumeration of numerical values, it is quite hard to identify their meaning on their own. Luckily, we can find the values and their corresponding names in the *Visual Studio* installation directory in one of the `msobj.lib` binaries. For example, Figure 3 shows the `msobj140-msvcrt.lib` file, which can be found in *Visual Studio 2015 – 2019*, and which contains a list of all `ProdID` values followed by their names.

The easiest way to find the Rich Header is to look for the `Rich` DWORD between the DOS and PE header (which marks the end of the Rich Header), extract the following XOR key DWORD and go backwards, XORing the data until the decrypted `DanS` DWORD is reached.



```
    prodidCvtomf511      = 0x0011,      // LINK 5.11 OMF to COFF conversion
    prodidMasm614        = 0x0012,      // MASM 6.14 (MMX2 support)
    prodidLinker512      = 0x0013,      // LINK 5.12
    prodidCvtomf512      = 0x0014,      // LINK 5.12 OMF to COFF conversion
#define DwProdidFromProdidWBuild(prodid, wBuild) ((((unsigned long) (prodid)) << 16) | (wBuild))
#define ProdidFromDwProdid(dwProdid)             ((PRODID) ((dwProdid) >> 16))
#define WBuildFromDwProdid(dwProdid)             ((dwProdid) & 0xFFFF)
    // Define the image data format
typedef struct PRODITEM {
    unsigned long   dwProdid;          // Product identity
    unsigned long   dwCount;           // Count of objects built with that product
} PRODITEM;

enum {
    tagEndID     = 0x536e6144,
    tagBegID     = 0x68636952,
};
/*
  Normally, the DOS header and PE header are contiguous.  We place some data
  in between them if we find at least one tagged object file.
  struct {
    IMAGE_DOS_HEADER dosHeader;
    BYTE             rgbDosStub[N];          // MS-DOS stub
    PRODITEM         { tagEndID, 0 };        // start of tallies (Masked with dwMask)
    PRODITEM         { 0, 0 };               // end of tallies   (Masked with dwMask)
    PRODITEM         rgproditem[];           // variable sized   (Masked with dwMask)
    PRODITEM         { tagBegID, dwMask };   // end of tallies
    PRODITEM         { 0, 0 };               // variable sized
    IMAGE_PE_HEADER  peHeader;
    };
```

Figure 2: Rich Headers definition in Microsoft Windows 2000 source code leak.



```
0028D990:  03 00 0A 00-70 72 6F 64-69 64 55 74-63 31 32 5F  ♥ ◙ prodidUtc12_
0028D9A0:  43 00 02 15-03 00 0B 00-70 72 6F 64-69 64 55 74  C ☻§♥ ♂ prodidUt
0028D9B0:  63 31 32 5F-43 50 50 00-F2 F1 02 15-03 00 0C 00  c12_CPP ≥±☻§♥ ♀
0028D9C0:  70 72 6F 64-69 64 41 6C-69 61 73 4F-62 6A 36 30  prodidAliasObj60
0028D9D0:  00 F1 02 15-03 00 0D 00-70 72 6F 64-69 64 56 69   ±☻§♥ ♪ prodidVi
0028D9E0:  73 75 61 6C-42 61 73 69-63 36 30 00-F2 F1 02 15  sualBasic60 ≥±☻§
0028D9F0:  03 00 0E 00-70 72 6F 64-69 64 4D 61-73 6D 36 31  ♥ ♫ prodidMasm61
0028DA00:  33 00 02 15-03 00 0F 00-70 72 6F 64-69 64 4D 61  3 ☻§♥ ☼ prodidMa
0028DA10:  73 6D 37 31-30 00 02 15-03 00 10 00-70 72 6F 64  sm710 ☻§♥ ► prod
0028DA20:  69 64 4C 69-6E 6B 65 72-35 31 31 00-F2 F1 02 15  idLinker511 ≥±☻§
0028DA30:  03 00 11 00-70 72 6F 64-69 64 43 76-74 6F 6D 66  ♥ ◄ prodidCvtomf
0028DA40:  35 31 31 00-F2 F1 02 15-03 00 12 00-70 72 6F 64  511 ≥±☻§♥ ↕ prod
0028DA50:  69 64 4D 61-73 6D 36 31-34 00 02 15-03 00 13 00  idMasm614 ☻§♥ ‼
0028DA60:  70 72 6F 64-69 64 4C 69-6E 6B 65 72-35 31 32 00  prodidLinker512
0028DA70:  F2 F1 02 15-03 00 14 00-70 72 6F 64-69 64 43 76  ≥±☻§♥ ¶ prodidCv
0028DA80:  74 6F 6D 66-35 31 32 00-F2 F1 02 15-03 00 15 00  tomf512 ≥±☻§♥ §
0028DA90:  70 72 6F 64-69 64 55 74-63 31 32 5F-43 5F 53 74  prodidUtc12_C_St
0028DAA0:  64 00 02 15-03 00 16 00-70 72 6F 64-69 64 55 74  d ☻§♥ ▬ prodidUt
0028DAB0:  63 31 32 5F-43 50 50 5F-53 74 64 00-F2 F1 02 15  c12_CPP_Std ≥±☻§

    prodidLinker512      = 0x0013,      // LINK 5.12
```

Figure 3: ProdID names as found in the msobj.lib binary.

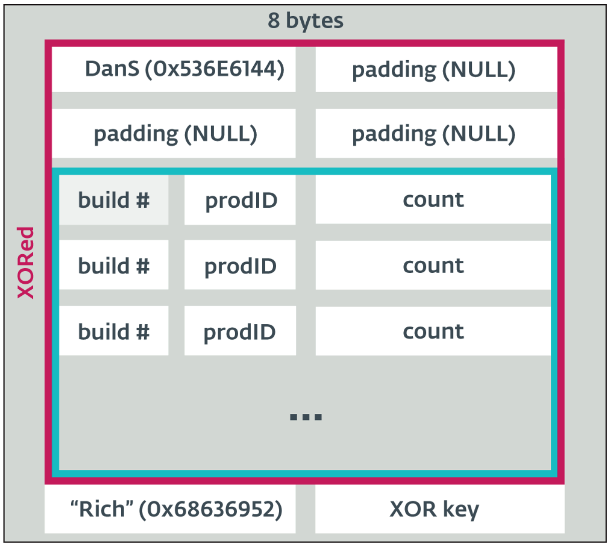
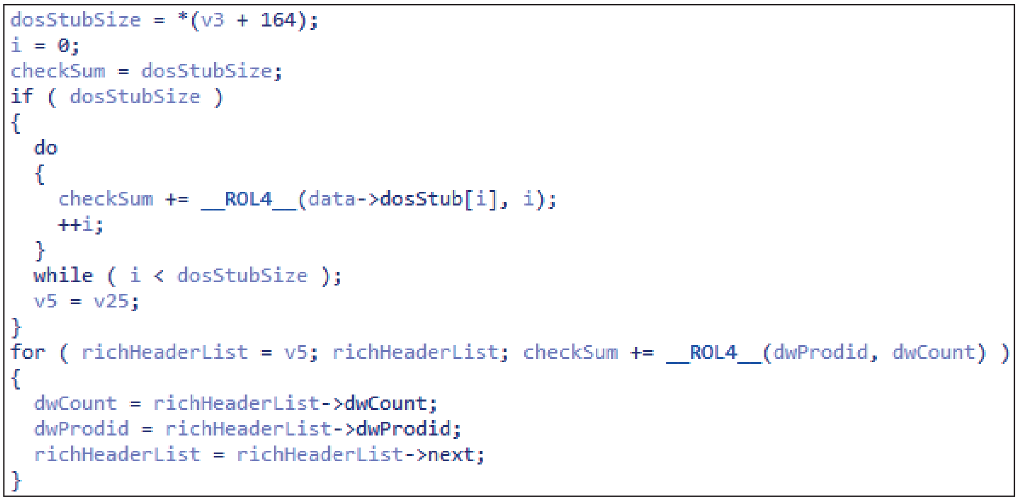


Figure 4: The unencrypted Rich Headers structure.

XOR key (checksum)

The XOR key used for encryption of the Rich Header is a unique four-byte value generated for every executable built by a *Microsoft* compiler (linker). The value is a checksum of the DOS header, the DOS stub and plaintext Rich Header data. The checksum calculation algorithm can be found in the `IMAGE::CbBuildProdidBlock` function in *Visual Studio*'s `link.exe` binary. A code snippet is shown in Figure 5.



```
dosStubSize = *(v3 + 164);
i = 0;
checkSum = dosStubSize;
if ( dosStubSize )
{
  do
  {
    checkSum += __ROL4__(data->dosStub[i], i);
    ++i;
  }
  while ( i < dosStubSize );
  v5 = v25;
}
for ( richHeaderList = v5; richHeaderList; checkSum += __ROL4__(dwProdid, dwCount) )
{
  dwCount = richHeaderList->dwCount;
  dwProdid = richHeaderList->dwProdid;
  richHeaderList = richHeaderList->next;
}
```

Figure 5: XOR key generation in Microsoft's Visual Studio 2019 linker executable.

Since the checksum algorithm consists of a simple addition and 32-bit rotate left, it can produce collisions. We have identified a few cases where the collision happens for one of two specific reasons.

The first case is when two projects contain identical `PRODITEM` entries, but in a different order. This is not necessarily unwanted behaviour and might even be intended by its designer. Both samples in Table 1 have the XOR key 0xAEB29219 [2].

kb.dll (2010-12-6 13:36:13)	phpF746.exe (2017-11-19 7:7:58)
prodidMasm614 | b8444 | 1	prodidLinker512 | b8078 | 9
prodidLinker512 | b8078 | 9	prodidMasm614 | b8444 | 1

Table 1: XOR key collision example – order of `PRODITEM`s.

Another common case is due to the 32-bit rotation used in the algorithm (`ProdID` + build is rotated by its count). If two projects have identical `ProdID` and build but different counts whose delta is a multiple of 32, the resulting checksum will be identical. Table 2 is another example of two samples that also have the XOR key 0xAEB29219.

moar.exe (2017-5-9 10:24:58)	phpF746.exe (2017-11-19 7:7:58)
prodidLinker512 | b8078 | 41	prodidLinker512 | b8078 | 9
prodidMasm614 | b8444 | 1	prodidMasm614 | b8444 | 1

Table 2: XOR key collision example – counts delta.

Behaviour

Rich Headers reflect the nature and scale of the project and simply by looking at the `ProdID` names, we can see the various features used in the project like imports, resources, or whether the project was written in Assembly, C, C++ or (pre .NET) Visual Basic, and approximately how large the project is, based on the product counts.

Rich Headers tend to change over time in an actively developed project – new `ProdID`s may appear and their counts often increase as new source files and other resources are added to the project, and with regular updates to *Visual Studio* the build numbers can change too. When changing a development environment completely (e.g. upgrading from *VS2010* to *VS2015*) `ProdID`s will also change for the equivalent in the new version.

Statistics

The most important question when talking about how useful Rich Headers may be is 'How often do they actually appear in executables?' If Rich Headers appeared only in a small portion of files, they would not be a very useful feature. Luckily, this is not the case: when looking at our large malicious dataset consisting of roughly one million unique malicious Win32 and Win64 native binaries (no .NET), we found that 73.20% of the PEs contain a Rich Header.

Of the 26.80% of files that do not contain a Rich Header, many are produced by other compilers (Turbo C++, MinGW GCC, Clang, etc.), or are developed in a completely different environment such as Delphi or Go. When such files are discarded, the percentage of PEs containing Rich Headers rises to 83.30%. At this point, the absence of a Rich Header is caused by explicit removal by malware authors or custom packers, etc. For our clean dataset of 120,000 samples, the representation goes as high as 94.20% and 98.50% when taking out other compilers.

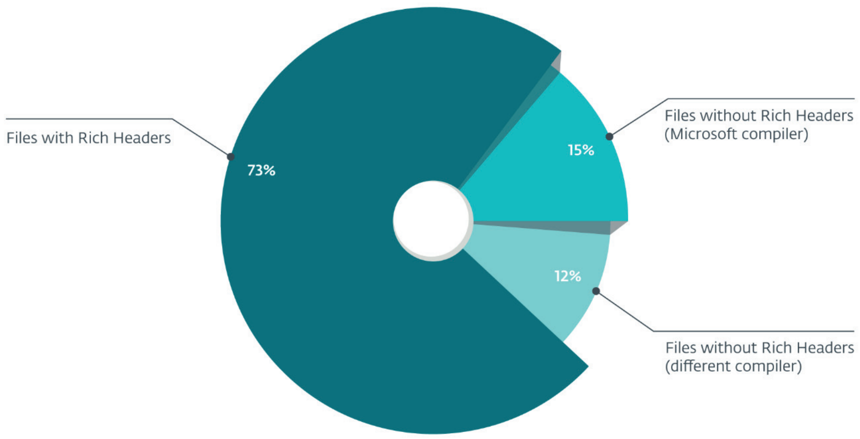


Figure 6: Representation of Rich Headers in our malicious dataset.

We expect this number to decrease slowly over time, as malware authors become more aware of this feature and will try to obfuscate or remove the header completely. A more rapid shift would happen if *Microsoft* decided to drop this feature completely in some subsequent version of *Visual Studio*, but as of the recent release of *Visual Studio 2019*, Rich Headers are still generated by the linker.

USE CASES

Levels of similarity

Identical Rich Headers

This is the strongest link between two files in terms of Rich Header similarity. Considering how often the Rich Headers may change in an actively developed project, it is not the best method of long-term detection of future samples from a given project (a good analogy is hash-based detection of malicious files). That being said, it can still be very handy in various cases.

The most common case is the simple matching of the Rich Headers of an unknown file to those of some known malware to quickly identify if the file is malicious (rather like looking up a hash on *VirusTotal*). Rich Headers can remain unchanged for weeks or even months in the development cycle of a specific project, so a complete match with older or even newer versions of the project can still happen if the sample set is good enough.

Another useful case is possible thanks to the fact that commercial protectors like *Enigma Protector*, *Themida* and *VMProtect* preserve the Rich Headers of the original file. This means that when we get our hands on, say, a *Themida*-protected file, we could potentially look up the original, non-packed payload by searching for identical Rich Headers and vice versa. This is especially valuable considering that most of these protectors change the nature of the file completely and there are no links preserved between the code or data of the original payload. Malware authors sometimes use multiple commercial packers to pack their payloads – we were able to identify the protected samples of a family called Predator Stealer [3] (Win32/Spy.Agent.{PQC,PQK}) and to attribute several custom Lazarus tools in [4] (Win64.NukeSped.{AA,AB}).

A similar strategy can also be applied to less advanced malware packers that also tend either to preserve the original Rich Header of the payload or to supplement the resulting packed file with their own.

Sometimes the Rich Header might intentionally be misplaced – there is a well-known case reported by *Kaspersky* [5] (and also presented at *Virus Bulletin* [6]), where a component of Olympic Destroyer contained a copy of the complete Rich Headers from a file previously attributed to the Lazarus toolset. This is widely considered to have been a false flag implemented by the attackers to place the blame on a different actor. On the other hand, it might very well have been an Easter egg, testing the attention of malware researchers. In the past there have also been cases of malware packers that copied the header from a legitimate `explorer.exe`.

For a quick matching, it is best to calculate and store a hash of the Rich Header. In our project we use an indexed CRC64 for a blazingly fast lookup.

Identical XOR keys

Matching XOR keys have very similar use cases to matching Rich Headers. The advantage of using the XOR key instead of a custom hash is in it being more widely used and being supported by widespread community tools like YARA, and integrated into *VirusTotal* hunting rules. The disadvantages lie in the fact that the value might artificially/explicitly be changed and in the potential key collisions that we explained in the previous section.

Knowing that the XOR keys are basically a checksum of the DOS header, DOS stub and Rich Header, we can quickly establish that, as a malware project evolves, the key value may change fairly often (with even a single incremental change in the Rich Header count), so it is not a very useful feature when trying to track a project over a longer period of time.

On the other hand, as stated above, a malware packer is a specific project and, in many cases, shares an identical Rich Header across many packed samples. We have found several values that covered reasonable clusters, e.g. 0x8F44CEBF, 0xAEB29219, 0x8A17753B, 0xD4F1AE19, 0x887F83A7 (with the CRC64 hash varying within each key!).

There are several XOR keys associated with well-known formats, as listed in Table 3. They can be found in clean as well as malicious datasets and might be useful for a quick identification of the format.

Identical ProdIDs + builds (unsorted)

Disregarding the counts and matching only the files with identical `ProdID`s and build numbers also turned out to be a very effective way of looking up similar files. The bigger the project, the more effective this method is, and it should be considered for samples with at least five Rich Header records.

In the case of the KillDisk-ed casino in Central America [7] we found a 64-bit executable with the internal name `res.dll`. Another file was later acquired, with all metadata pointing to the same threat actor. Despite being Themida-protected and statically completely different (except for the agreement of PE timestamps), the set of `ProdID`s together with builds were identical and did not match any

Visual Basic 6.0	0x886973F3, 0x8869808D, 0x88AA42CF, 0x88AA2A9D, 0x89A99A19, 0x88CECC0B, 0x8897EBCB, 0xAC72CCFA, 0x1AAAA993, 0xD05FECFB, 0x183A2CFD, 0xACCF9994, 0xC757AD0B, 0xA7EEAD02, 0xD1197995, 0x83CDAD4, 0x8917A389, 0x88CEA841, 0x8917DE83, 0x89AA0373, 0x8ACD8739, 0x8D156179, 0x8ACE4D53, 0x8897FE31, 0x91A515F9, 0xD1983193, 0x8D16E113, 0x9AC47EF9, 0x91A80893, 0xAD0350F9, 0xD180F4F9, 0xAD0EF593, 0x9ACA5793, 0x9ACA5793
NSIS	0xD28650E9, 0x38BF1A05, 0x6A2AD175, 0xD246D0E9, 0x371742A2, 0xAB930178, 0x69EAD975, 0x69EB1175, 0xFB2414A1, 0xFB240DA1
MASM 6.14 build 8444	0x88737619, 0x89A56EF9
WinRar SFX	0xC47CACAA, 0xFDAFBB1F, 0xD3254748, 0x557B8C97, 0x8DEFA739, 0x723F06DE, 0x16614BC7
Autoit	0xBEAFE369, 0xC1FC1252, 0xCDA605B9, 0xA9CBC717, 0x8FEDAD28, 0x273B0B7D, 0xECFA7F86
Microsoft Cabinet File	0x43FACBB6
NTkernelPacker	0x377824C3
Thinstall	0x8B6DF331
MoleBox Ultra v4	0x8CABE24D

Table 3: Various XOR keys associated with known formats.

clean files – see Table 4 (both detected as Win64/NukeSped.Z). This inspired us to index the sequence of pairs (`ProdID`, build) as they are located in the header, omitting the counts. The CRC64 checksum of unsorted sequences of these pairs reliably identified several clusters of related malicious projects: not only when the single count of `prodidImports0` varied (Win64/CoinMiner.DN, 0x105E60A5B349F444), but also when multiple counts did (Win32/Pterodo, 0x745E73E5045EE80E or Win32/Kryptik.GTXI, 0x613D0D87DAF1658F).

rds.dll (2016-10-24 8:32:11)	res.dll (2016-10-24 8:32:11)
prodidUtc1600_LTCG_CPP| b40219 | 9	prodidUtc1600_LTCG_CPP| b40219 | 9
prodidUtc1600_CPP | b40219 | 27	prodidUtc1600_CPP | b40219 | 27
prodidUtc1600_C | b40219 | 100	prodidUtc1600_C | b40219 | 94
prodidMasm1000 | b40219 | 15	prodidMasm1000 | b40219 | 9
prodidLinker1000 | b40219 | 1	prodidLinker1000 | b40219 | 1
prodidExport1000 | b40219 | 1	prodidExport1000 | b40219 | 1
prodidImplib900 | b30729 | 9	prodidImplib900 | b30729 | 9
prodidImport0 | b0 | 130	prodidImport0 | b0 | 131

Table 4: Similar metadata from two Themida-protected 64-bit Lazarus backdoors.

Identical ProdIDs + builds (sorted)

The situation gets more complicated when searching for files from the same project that have different architectures. Rich Headers tend to be almost identical between 32-bit and 64-bit equivalents of an application, but sometimes the `ProdID`s are in a slightly different order. This led us to a new similarity level, namely the CRC64 checksum of the sequence of `ProdID`s and their builds sorted by the `ProdID` values to compare the Rich Header regardless of the order. Again, the counts were omitted.

This turned out to be useful in many cases, for example when we encountered several almost identical samples of Win/GreyEnergy [8], each for a different architecture, having the same `ProdID`s, but in a slightly different order.

zlib_x86.dll, zlibwapi_x86.dll			zlib_x64.dll, zlibwapi_x64.dll		
prodidAliasObj1000	b20115	5	prodidAliasObj1000	b20115	5
prodidUtc1600_CPP	b40219	25	prodidUtc1600_CPP	b40219	25
prodidMasm1000	b40219	17	prodidUtc1600_C	b40219	107
prodidUtc1600_C	b40219	110	prodidMasm1000	b40219	9
prodidImplib900	b30729	3	ProdidImplib900	b30729	3
prodidImport0	b0	[87-88]	prodidImport0	b0	[88-89]
prodidUtc1600_LTCG_CPP	b40219	22	prodidUtc1600_LTCG_CPP	b40219	[22-23]
prodidExport1000	b40219	1	prodidExport1000	b40219	1
prodidLinker1000	b40219	1	prodidLinker1000	b40219	1

Table 5: Similar metadata from Win32/GreyEnergy payloads for both platforms.

Specific values query/conjunction of various ProdIDs

While matching identical Rich Headers is nice and can be useful in many cases, for a more robust lookup we need some variability. This can be achieved by searching for combinations of specific `ProdID`s combined with builds and a range of counts based on a value from the underlying sample. If such a query is constructed in a smart way, it can easily yield many older and/or newer samples from the same project with slightly different Rich Headers as development progressed. For example, given a file that contains `prodidUtc1600_CPP` build 40219 with a count of 48, we might search for all files with `prodidUtc1600_CPP` build 40219 and a count between 42 and 54. So far we have not found a generic approach for how best to construct these queries and instead use our 'instincts' that we have acquired over time while working with Rich Headers.

For example, there were cases of a two-stage threat when the dropper and the payload were projects of very similar size, e.g. Win32/Exaramel [9]. Both files were 32-bit, so the order of `ProdID`s was identical and counts very similar, with the exception of one additional `ProdID` in the payload, as seen in Table 6.

xml.exe (2018-4-13 08:36:04)	wsmprovav.exe (2018-4-13 08:35:52)
prodidUtc1900_CPP | b24215 | 3	prodidUtc1900_CPP | b24215 | 1
prodidUtc1900_CPP | b24123 | 33	prodidUtc1900_CPP | b24123 | 29
prodidUtc1900_C | b24215 | 7	prodidUtc1900_C | b24215 | 11
prodidUtc1900_C | b24123 | 17	prodidUtc1900_C | b24123 | 17
prodidMasm1400 | b24123 | 18	prodidMasm1400 | b24123 | 17
prodidLinker1400 | b24215 | 1	prodidLinker1400 | b24215 | 1
prodidCvtres1400 | b24210 | 1	prodidCvtres1400 | b24210 | 1
prodidUtc1810_CPP | b40116 | 121	prodidUtc1810_CPP | b40116 | 120
prodidUtc1810_C | b40116 | 24	prodidUtc1810_C | b40116 | 24
prodidMasm1210 | b40116 | 9	prodidMasm1210 | b40116 | 9
prodidResource | b0 | 1	prodidResource | b0 | 1
prodidImplib900 | b30729 | 9	prodidImplib900 | b30729 | 25
prodidImport0 | b0 | 109	prodidUtc1500_C | b30729 | 4
	prodidImport0 | b0 |200

Table 6: Similar metadata for the dropper and the payload of Win32/Exaramel.

One should always try to keep the number of items in the query to a bare minimum to distinguish different projects, but at the same time leave some space for potential changes. Generally, `ProdIDs` containing '_CPP' or '_C' should be prioritized in the search as they tend to have the biggest count variability and the count distance could be around +/- 10%.

Anomalies

Basing detection on Rich Header similarity or adding it as a feature to existing detection systems is a great thing. Another potential use case is heuristic detection based on anomalies and inconsistencies that suggest the file has been tampered with, or that there is something wrong with it, and that should generally be brought to the attention of a researcher. Such an anomaly obviously does not instantly mean a given file is malicious, but it might be useful to flag files worthy of closer examination by an analyst. Anomalies are especially common in relation to various packers and other obfuscation strategies commonly found in malware, as they are usually the result of tampering with the original file.

In the following subsections we look at the most common anomalies we have observed in the wild.

Invalid Rich Header values

This is the most obvious anomaly – since the range of valid `ProdIDs` is known, we can easily identify files that have non-existent values (at the moment, with the release of *Visual Studio 2019*, all values larger than 0x010e (`prodidUtc1900_POGO_O_CPP`) can be considered invalid). This could be applied to the build number as well, as each `ProdID` has only a specific range of valid build numbers, although this would require quite a lot of work due to having to gather all possible build numbers for each `ProdID`. Finally, we could also check for suspicious count values, as for valid projects they are generally in the tens or hundreds and very rarely in the thousands, so finding a valid `ProdID` with a count greater than a million could be an indicator too.

Duplicate Rich Header values

Every file with a Rich Header can only have a single, unique combination of `ProdID` and a build number. This is obvious, as the number of those records is represented by the count field instead, so duplicate records are invalid behaviour. Around 1.19% of files in our malicious dataset contain duplicate records. There is not a single one in the clean dataset.

Invalid XOR key checksum

Since we know how the checksum is calculated, the supplied XOR key can easily be verified. If the value is incorrect, we can assume that there is potentially something wrong. In our malicious dataset 9.06% files have an invalid XOR key value, while in the clean dataset only around 0.95% do.

Rich Header offset

The Rich Header offset is the position in the PE at which the header begins. It is based on the size of the DOS stub, as it begins right after that, so its value varies with the DOS stub. Since most files nowadays do not use the DOS part of the PE at all, they have the widely known default '*This program cannot be run in DOS mode*' stub generated by the compiler, and thanks to that, in most files the Rich Header starts at file offset 0x80. While the mentioned starting position is present in ~99% of cases, there are many, even valid, cases where the offset differs, so it is not a good idea to look for Rich Headers based on this offset (like YARA does [10]). Out of 120,000 real-world clean applications with Rich Headers in our clean dataset, only 0.06% do not match the offset 0x80. The distribution for our malicious dataset can be seen in Figure 7.

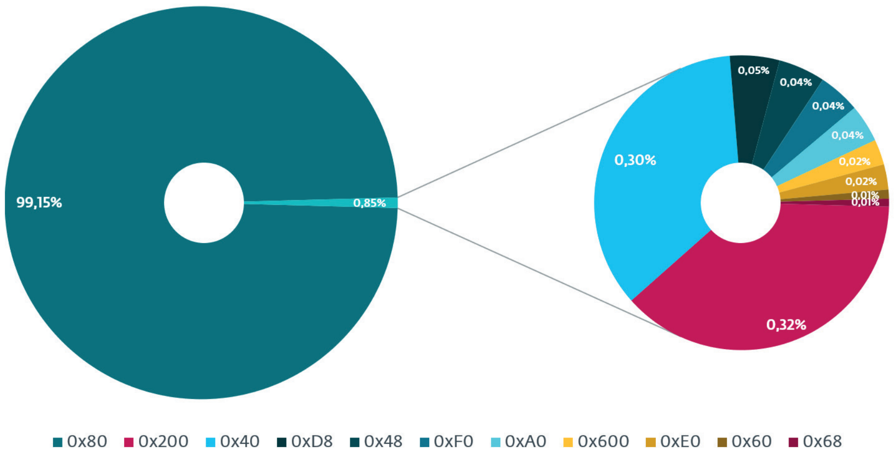


Figure 7: Distribution of Rich Header offsets in our malicious dataset.

There are a few relatively common offsets that can be found even in clean, legitimate projects like some kernel drivers (probably caused by an alignment issue, because even the default stub is aligned to 512 bytes). On the other hand, there are also several offsets that are used by a specific malware family or software project (those we have identified are listed in Table 7), which could be one of the ways to identify new samples or at least used as an indicator.

Offset	**Example**	**Comment**
0x40	Various	The null DOS stub
0x48	Various	The minimal valid DOS stub
0x60	Win32/Adware.WintionalityChecker	
0x68	Win32/TrojanDownloader.Waski	
0x8C	Win32/Adware.Virtumonde	
0xE0	Win32/TrojanDropper.Agent.NJV	
0x100	ESET modules	Clean
0x200	Various	Some kernel drivers
0x2A8	PKSFX Self Extract Utility by PKWARE Inc.	Clean
0x600	Win32/Adware.Trymedia	

Table 7: Various types of Rich Header offsets.

PE Optional Header linker information mismatch

The version of a linker used for building the file is also stored in the `MajorLinkerVersion` and `MinorLinkerVersion` fields inside the PE Optional Header. Comparing this version to the linker version found in Rich Headers (i.e. `prodidLinker800` should have Optional Header linker version set to 8.0) is also a good way to find inconsistencies, packed files, tampered Rich Headers and so on.

There are a few such anomalies caused by known packers – for example, the Armadillo packer changes the Optional Header linker version to 83.82, but does not alter the Rich Header.

Imports

The `prodidImport0` identifier is related to the number of functions in the PE import directory. The values are not always equal but correlate highly and tend to be very close together. A large difference between the number of imports and the count of `prodidImport0`, or its complete absence in the Rich Header when imports are present, and vice versa, could be a sign of a packed file or a faked Rich Header.

Resources

When an executable contains a resource directory populated with resources, it should always be reflected in the Rich Header by containing `prodidCvtresXXXX`, where XXXX is the version of *Visual Studio* used. The count should always be 1, regardless of the number of resources the file has.

For example, when building a project that has any kind of resource included in *Visual Studio 2015* (14.0), the resulting binary should always contain `prodidCvtres1400` = 1 (in our malicious dataset, 4.12% of files have a `prodidCvtresXXXX` count other than 1; in our clean dataset this applies to only 0.01% of files).

There are multiple ideas for potential heuristics:

1. Files with `prodidCvtresXXXX` having a count not equal to 1
2. Files with resources not containing the `Cvtres` identifier and vice versa
3. The version of `prodidCvtresXXXX` not corresponding with the linker version/other `ProdIDs`, i.e. the file having `prodidCvtres1400` and `prodidLinker1200`.

APT malware

In this section we provide some examples where Rich Header metadata has provided interesting information and extended our overall knowledge about executables from APT toolsets. This is not the first attempt to look at Rich Headers and their usability in APT research [11].

Win32/Industroyer & Win32/Exaramel

The article [9] uncovered strong code similarities between the Win32/Industroyer main backdoor (introduced to the world in [12]) and one of the backdoors (Win32/Exaramel) used by TeleBots, the group behind the massive NotPetya (Win32/Diskcoder.C) ransomware outbreak. Taking an additional look at the Rich Headers metadata, it turns out that there is additional similarity at the project development and build level – see Table 8.

prodidMasm1210	b40116	=	9
prodidUtc1810_CPP	b40116	=	120
prodidUtc1810_C	b40116	=	24
prodidMasm1400	b24123	=	17
prodidUtc1900_CPP	b24123	=	29
prodidUtc1900_C	b24123	=	17
prodidImplib900	b30729	=	[15-25]
prodidImport0	b0	=	[140-200]
prodidCvtres1400	b24210	=	1
prodidLinker1400	b24215	=	1

Table 8: Rule based on the Rich Header values common to Industroyer and Exaramel.

Win{32,64}/NukeSped

This is *ESET*'s detection name for samples from the Lazarus toolset. It is well known that the toolset is large, heterogeneous from various points of view, and all the sources are kept close. In [4] we tried to group executables based on their Rich Header characteristics and the version of the *Microsoft* linker used. As a result, two main Lazarus tool subgroups were identified, namely the first one using *VS 98* for

32-bit compilations and *VS 2010* for 64-bit, and the second one using *VS 2010* for both. However, several anomalous combinations were discovered too, e.g. one typical malicious project compiled as a 32-bit binary in *VS 98* and as a 64-bit binary in *VS 2013*. In Table 9 there are two examples of queries that successfully identified malicious clusters without causing an uncomfortable level of noise.

```
prodidUtc1600_C      b40219 = [50-300]
prodidImports        b0     = [227-228]
prodidImplib900      b30729 = 23
prodidAliasObj1000   b20115 = 7

prodidUtc1500_C      b30729 = 8
prodidExport1000     b40219 = 1
prodidUtc1600_C      b40219 = [142-145]
prodidUtc1600_CPP    b40219 = [61-62]
```

Table 9: Win{32,64}/NukeSped detection rule.

Win32/CrisisHT

This is *ESET*'s detection name for samples from the spyware kit operated and sold by the infamous Italian software company Hacking Team. In [13] many aspects of the threat are considered, e.g. the clustering of the toolset based on the attached code-signing certificates. In Table 10 we illustrate clustering based on the Rich Header values. The 'SPC' version looks like a testing concept built by an unrelated team – a consequence of the source code leaks in July 2015.

Certificate	**PE Timestamp**	**`prodidUtcXXXX_CPP` and build values**					**Import0**
		1600_CPP	**1600_CPP**	**1700_CPP**	**1800_CPP**	**1900_CPP**	
		b40219	**b30319**	**b50929**	**b21005**	**b24123**	
Raffaele Carnacina	16.9.2015		38				170
Raffaele Carnacina	19.10.2015		64				247
Raffaele Carnacina	19.10.2015		38				169
Raffaele Carnacina	5.11.2015	38					168
Raffaele Carnacina	17.11.2015	64					247
SPC	5.1.2016				49		163
Raffaele Carnacina	18.1.2016	38					170

Table 10: CrisisHT samples clustered by C++ compiler version Rich Header data.

Certificate	PE Timestamp	`prodidUtcXXXX_CPP` and build values					Import0
		1600_CPP b40219	1600_CPP b30319	1700_CPP b50929	1800_CPP b21005	1900_CPP b24123	
Raffaele Carnacina	24.3.2016		64				247
Raffaele Carnacina	24.3.2016		38				169
ADD Audit	17.6.2016			70			251
ADD Audit	1.9.2016			70			250
ADD Audit	19.12.2016			44			177
ADD Audit	28.4.2017	38					170
ADD Audit	28.4.2017	64					259
Ziber Ltd	28.6.2017					57	293
Media Lid	28.6.2017	40					184
Ziber Ltd	9.10.2017	40					190
Ziber Ltd	18.10.2017					57	299

Table 10 (contd.): CrisisHT samples clustered by C++ compiler version Rich Header data.

Crimeware

Win32/Kasidet

Kasidet (a.k.a. Neutrino bot) is a crime kit with many capabilities which is sold on underground forums and is popular for its relatively cheap price. The crime kit is distributed via a special builder containing the compiled stub of Neutrino bot. In Table 11 we showcase the evolution of four `ProdIDs` that represent the projects. Each colour band represents a unique stub that was distributed by various cybercriminal groups [14].

	PE timestamp	prodidUtc1500_C	prodidImplib900	prodidImport0	prodidUtc1600_CPP
MiningRats 04	25.07.2017	1	25	175	44
LethicGuys 08	21.10.2017	1	29	179	46
Redirectors 01	25.10.2017	1	29	179	46
LethicGuys 09	31.10.2017	1	29	179	47
LethicGuys 10	07.11.2017	4	31	185	49
Redirectors 03	07.11.2017	4	31	185	49
LethicGuys 11	16.11.2017	4	31	185	49
Redirectors 04	16.11.2017	4	31	186	49
Redirectors 05	17.11.2017	4	31	186	49
MiningRats 07	05.01.2018	4	31	186	49
MiningRats 08	08.01.2018	4	31	186	49
MiningRats 09	12.01.2018	3	31	196	50
LethicGuys 13	13.01.2018	3	31	196	50
LethicGuys 14	15.01.2018	3	31	196	50
ProxyGuys 01	16.01.2018	3	31	196	50
MiningRats 10	20.01.2018	3	31	196	50
LethicGuys 15	21.01.2018	3	31	196	50
LethicGuys 16	21.01.2018	3	31	196	50
MiningRats 11	30.01.2018	3	29	193	49
ProxyGuys 02	04.02.2018	3	29	193	49
TinukeFareit 01	09.02.2018	3	29	193	49
LethicGuys 17	16.02.2018	3	29	193	49
LethicGuys 18	22.02.2018	3	29	193	49
LethicGuys 19	25.02.2018	3	29	193	49
LethicGuys 20	26.02.2018	3	29	193	49
Redirectors 06	16.03.2018	3	29	193	49
Redirectors 07	19.03.2018	3	29	193	49
LethicGuys 21	22.03.2018	3	29	193	49

Table 11: The evolution of the Win32/Kasidet project.

	PE timestamp	prodidUtc1500_C	prodidImplib900	prodidImport0	prodidUtc1600_CPP
LethicGuys 22	22.03.2018	3	29	193	49
LethicGuys 23	02.04.2018	3	29	194	49
ProxyGuys 03	02.04.2018	3	29	194	49
LethicGuys 24	05.04.2018	3	29	194	49
TinukeFareit 02	08.04.2018	3	29	194	49
TinukeFareit 03	23.04.2018	3	29	194	49
TinukeFareit 04	02.05.2018	3	29	194	49
Arsonists 01	07.05.2018	3	29	194	49
TinukeFareit 05	19.05.2018	3	27	192	49
Arsonists 021	21.05.2018	3	27	192	49
TinukeFareit 061	21.05.2018	3	27	192	49
LethicGuys 25	23.05.2018	3	27	192	49
ProxyGuys 04	06.06.2018	3	29	223	54
Arsonists 022	10.06.2018	3	29	223	49
LethicGuys 26	14.06.2018	3	29	224	54
TinukeFareit 062	19.06.2018	3	29	224	55
TinukeFareit 07	05.07.2018	3	29	224	55
MiningRats 12	26.08.2018	3	29	241	56
LethicGuys 27	03.09.2018	3	29	241	56
Arsonists 03	18.09.2018	3	29	241	56
LethicGuys 28	18.09.2018	3	29	241	56
ProxyGuys 05	18.09.2018	3	29	241	56
TinukeFareit 08	20.09.2018	3	29	241	56
MiningRats 13	11.10.2018	3	29	241	56

Table 11 (contd.): The evolution of the Win32/Kasidet project.

Table 12 lists the Rich Header search rules we use to cover the prevalent Neutrino bot version 5.4.

```
prodidUtc1500_C   b30729 = [1-4]
prodidUtc1500_CPP b30729 = 1
prodidImplib900   b30729 = [25-31]
prodidimport0     b0     = [175-241]
prodidUtc1600_C   b40219 = 5
prodidUtc1600_CPP b40219 = [44-56]
```

Table 12: Win32/Kasidet version 5.4 detection rule.

Win{32,64}/Dridex

It seems that, at any given time, this infamous banking trojan is actively being developed by a single entity, as all the samples from a given time period/campaign have the same Rich Headers. The development environment has changed at least four times. This could mean that the authors updated their tools or that the source code has changed hands and has either been sold or passed to someone else. The detection rules for various versions can be found in Table 13.

Win{32,64}/Dridex v1, v2 & v3 (2014–2015)

The initial versions of the Dridex banking trojan were developed in *Visual Studio 2010*. The rule should cover all samples from version 1.100 all the way until 3.102, which is over a year-long period between July 2014 and September 2015. We have collected 75 unique builds based on this search and, based on the gaps in the versions of samples we were missing, it could potentially cover over 270 unique versions.

Win{32,64}/Dridex v3 (2016)

After version 3.102 the development environment changed slightly for the rest of the v3 development cycle and both 32-bit and 64-bit versions between v3.145 (November 2015) and v3.258 (September 2016) can be covered by the rule.

Win{32,64}/Dridex v4 (2017–2018)

Dridex version 4 introduced a few major changes including the infamous Atom bombing technique that made the headlines back in 2017 as well as a major development environment towards *Visual Studio 2015*. The rule covers the versions between v4.43 and v4.87 and reliably covers both 32-bit and 64-bit samples.

Win{32,64}/Dridex v5(v2) (2018–2019)

Another shift in the development environment of this malware. This time there has been an interesting shift in the versioning as well – instead of an expected version 5 it has been set back to version 2. This could indicate a change in the ownership of the source code or a change of the developer responsible for the malware. The rule should cover the most recent versions (v2.68 and newer).

Versions 1, 2 and 3 (2014–2015)	`prodidUtc1500_CPP b30729 = 1`
	`prodidMasm1000    b40219 = [23-26]`
	`prodidUtc1600_C   b40219 = [99-143]`
	`prodidUtc1600_CPP b40219 = [32-48]`
	`prodidUnknown     b0     = 1`

Version 3 (2016)	`prodidImplib900   b30729 = [3-5]`
	`prodidImport0     b0     = [1-2]`
	`prodidUtc1500_C   b30729 = 1`
	`prodidExport1000  b40219 = 1`
	`prodidUnknown     b0     = 1`

Version 4 (2017–2018)	`prodidUtc1900_C   b24215 = [22-23]`
	`prodidUtc1900_CPP b24215 = [58-60]`
	`prodidLinker1400  b24215 = 1`

Version 5 (v2) (2018–2019)	`prodidUtc1500_C   b30729 = 4`
	`prodidImplib900   b30729 = 5`
	`prodidImport0     b0     = 6`
	`prodidLinker1400  b24215 = 1`
	`prodidUnknown     b0     = 90`

Table 13: Win{32,64}/Dridex detection rules.

Win{32,64}/Sathurbot

This name represents botnets spreading via password attacks on *WordPress* sites [15]. The malware is a large project which is compiled for both 32- and 64-bit *Windows* platforms. Table 14 shows the evolution of the build environment (internal DLL name, PE Timestamp and `ProdIDs` representing the source files).

It is not possible to conclude with high confidence that there is more than a single entity operating the source code. However, the produced executables suggest that *Visual Studio* is switched regularly. The merged PE timestamp of two samples into one cell with a single date means a consecutive compilation for both 32- and 64-bit platforms that differed by just a few seconds. The order of this compilation varied as well.

Malware packers

The main purpose of malware packers is to provide a container for malware families with features like polymorphism, anti-analysis and obfuscation. Because of these features, malware packers often display abnormal static properties and their Rich Headers are no exception.

The first interesting case was a validly signed, highly polymorphic obfuscator detected as Win32/Kryptik.GSLI. In May 2019, hundreds of unique samples were distributed in the wild, with Win32/TrojanDownloader.Zurgop.CY as the protected payload. We discovered these campaigns because the `prodidCvtres500` identifier contained anomalously high counts. After collecting many related files, we found that the rules in Table 15 identified the cluster very well.

		prodidUtcXXXX_CPP				
	PE TS	1600_ CPP b40219	1600_ CPP b30319	1600_ LTCG_CPP b30319	1700_ CPP b50929	1800_CPP b20806
PerformanceMonitor.dll	22.4.2015				59	
PerformanceMonitor.dll	27.4.2015				60	
PerformanceMonitor.dll					59	
PerformanceMonitor.dll	14.5.2015				60	
PerformanceMonitor.dll					59	
PerformanceMonitor.dll	21.6.2015		92			
PerformanceMonitor.dll			91			
PerformanceMonitor.dll	8.7.2015		92			
PerformanceMonitor.dll	30.7.2015		91			
PerformanceMonitor.dll	4.8.2015		91			
PerformanceMonitor.dll	18.8.2015		89			
PerformanceMonitor.dll			88			
PerformanceMonitor.dll	27.8.2015		88			
PerformanceMonitor.dll	15.9.2015		89			
PerformanceMonitor.dll	8.2.2016	89				
PerformanceMonitor.dll	18.2.2016	89				
performanceMonitor_64.dll	31.3.2016					84
performanceMonitor_32.dll	29.06.2016					83
performanceMonitor_64.dll						84
performanceMonitor_32.dll	31.8.2016					83
performanceMonitor_32.dll	24.10.2016					83
performanceMonitor_64.dll	29.12.2016		69	97		
performanceMonitor_32.dll	13.1.2017		68	97		
performanceMonitor_32.dll	16.2.2017		68	97		
performanceMonitor_64.dll			69	97		
performanceMonitor_64.dll	10.3.2017		69	97		
performanceMonitor_64.dll	21.3.2017		69	97		
performanceMonitor_64.dll	27.5.2017					79
performanceMonitor_64.dll	7.11.2017					79

Table 14: The evolution of Win{32,64}/Sathurbot.

prodidMasm613	b7299	= [12-13]
prodidUtc12_C	b9782	= 63
prodidLinker512	b8034	= 11
prodidUtc12_CPP	b9782	= 2

Table 15: Detection query for Win32/Kryptik.GSLI.

Later, we identified another extremely polymorphic malware packer. The shared static properties include two `.version` PE sections of 1,024 bytes each, a file version of 1.0.0.1 and legal copyright starting with `Copyright (C) 2018`. The rest of the features seem randomly generated. We described the cluster with the relatively simple rule conjunction seen in Table 16.

prodidUtc1500_CPP	b21022	= 37
prodidMasm900	b21022	= 17
prodidUtc1500_C	b21022	= [112-120]
prodidUtc1500_LTCG_CPP	b21022	= 1
prodidCvtres900	b21022	= 1

Table 16: Detection query for the malware packer shared across many malware families.

There were many malware families protected with this packer, including the following (for further examples see [01]):

- Win32/PSW.Delf.OSF (aka Azorult)
- Win/Spy.Kronosbot
- Win32/TrojanDownloader.Zurgop.DA
- Win32/Filecoder.Spora
- Win32/Agent.TFY (a backdoor written in *GoLang*)
- MSIL/Spy.Baldr
- Win32/Ramnit.BV (a virus)
- Win32/Spy.Ursnif.CY

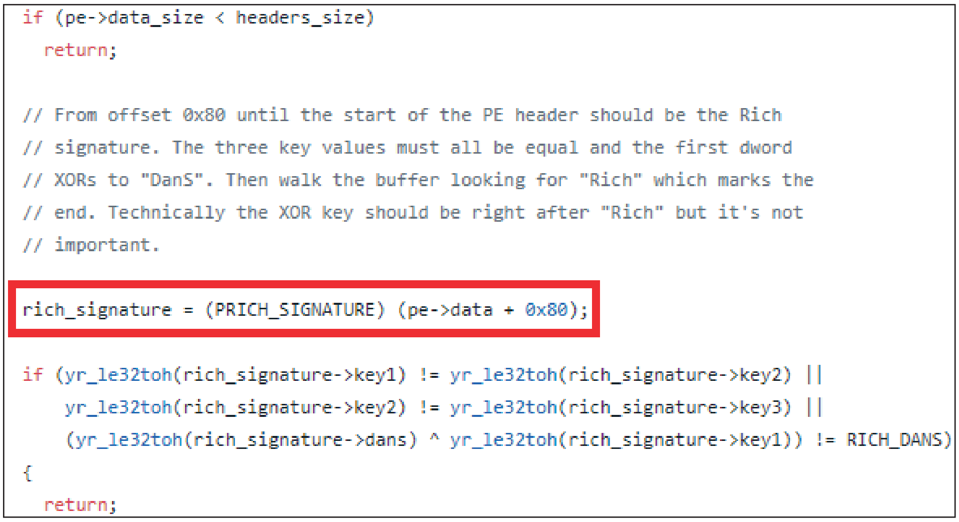
Rich Headers & YARA

YARA is a great tool for malware researchers, incident responders and security engineers that has become an industry standard for hunting malware and the classification of unknown samples. Hence, being able to write YARA rules based on Rich Headers would be a great way to share our findings with the rest of the world.

Although YARA does support parsing of Rich Headers and the PE module has a few Rich Header-related fields, at the time of writing (June 2019) there are a few critical drawbacks that greatly limit its usability [16].

First, as seen in its source code in Figure 8, YARA incorrectly assumes that the Rich Header always starts at the common offset 0x80 [10] and will not work for any Rich Header at a different offset.

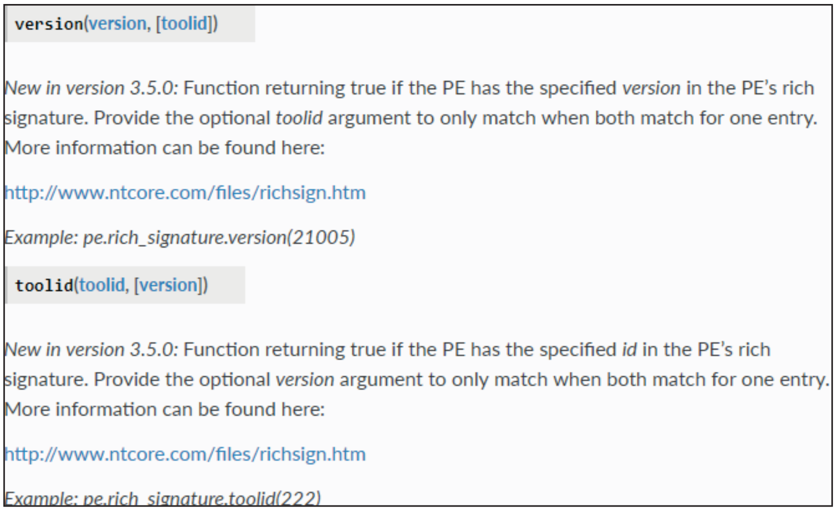
This can easily be fixed and is not that big an issue, since as stated above, over 99% of files with Rich Headers tend to have them at that offset.



```
if (pe->data_size < headers_size)
  return;

// From offset 0x80 until the start of the PE header should be the Rich
// signature. The three key values must all be equal and the first dword
// XORs to "DanS". Then walk the buffer looking for "Rich" which marks the
// end. Technically the XOR key should be right after "Rich" but it's not
// important.

rich_signature = (PRICH_SIGNATURE) (pe->data + 0x80);

if (yr_le32toh(rich_signature->key1) != yr_le32toh(rich_signature->key2) ||
    yr_le32toh(rich_signature->key2) != yr_le32toh(rich_signature->key3) ||
    (yr_le32toh(rich_signature->dans) ^ yr_le32toh(rich_signature->key1)) != RICH_DANS)
{
  return;
```

Figure 8: YARA's implementation of Rich Headers lookup.

The bigger problem is the YARA Rich Header-related API, which is very simple and insufficient for any effective Rich Header-related rules. The `version` and `toolid` functions can only look whether a specific `ProdID` and build is present in the Rich Header, but are unable to query its count values, which are the most crucial attribute in most useful searches. The announcement of YARA's `toolid` function can be seen in Figure 9.



version(version, [toolid])

New in version 3.5.0: Function returning true if the PE has the specified *version* in the PE's rich signature. Provide the optional *toolid* argument to only match when both match for one entry. More information can be found here:

http://www.ntcore.com/files/richsign.htm

Example: pe.rich_signature.version(21005)

toolid(toolid, [version])

New in version 3.5.0: Function returning true if the PE has the specified *id* in the PE's rich signature. Provide the optional *version* argument to only match when both match for one entry. More information can be found here:

http://www.ntcore.com/files/richsign.htm

Example: pe.rich_signature.toolid(222)

Figure 9: YARA PE module Rich Headers API documentation.

The APIs would be much more useful if instead of the existence of a given `ProdID`, the `toolid` and `version` interfaces returned the count (or null), which then could be compared against a ranged interval. This would greatly enhance YARA's Rich Header-related capabilities and would enable more complete leverage of the Rich Header format. We hope to contribute to the YARA project and to help to improve the Rich Header-related functionality in the near future.

CONCLUSION

We started our project with low expectations, thinking that there must be a reason the Rich Headers feature is overlooked and not widely utilized. Over time, we became more and more impressed with how much could be achieved by searching for feature clusters based on such a small part of an executable, and how powerful it can be when leveraged correctly.

The space of Rich Header features has an extremely fast lookup and is 'rich' indeed, in the sense that the detection queries for chosen parts of APT toolsets or crimeware kits identified the clusters well and with a low level of noise. Because of the nature of the information, false positives may occur, however the erroneous verdicts seem to be intuitive: clean projects of small magnitude, shared executable containers and, surprisingly, different malicious families as well. We believe this kind of static feature may also be utilized in machine-learning algorithms, to assist in deciding maliciousness, and this is the subject of future work.

Sharing publicly the various techniques malware researchers use to track cybercriminal actors usually causes an obvious reaction: avoidance. In the case of the Rich Header that may be a Pyrrhic victory – any interference with this delicate structure, such as wiping it or substituting it, will produce obvious, easily detectable anomalies just asking for closer attention from our malware analysts.

Rich Headers are definitely not a silver bullet that will change the security industry, but they could be a nice and very helpful addition to current detection and hunting methods and systems. We feel that this undocumented structure deserves more attention than it currently receives across the field and imagine that, in the not too distant future, we may see Rich Header YARA rules appearing in malware analysis IoC listings and the like.

REFERENCES

[1] https://github.com/AyalaRs/win/blob/master/private/sdktools/vctools/langapi/include/prodids.h.

[2] ESET GitHub. SHA256 hashes of mentioned files. https://github.com/eset/malware-ioc/tree/master/rich_headers.

[3] GReAT. A predatory tale: Who's afraid of the thief? Securelist.com. March 2019 https://securelist.com/a-predatory-tale/89779/.

[4] Kálnai, P.; Poslušný, M. Lazarus Group: a mahjong game played with different sets of tiles, Virus Bulletin International Conference 2018, Montreal. https://www.virusbulletin.com/uploads/pdf/magazine/2018/VB2018-Kalnai-Poslusny.pdf.

[5] Kaspersky GReAT. The devil's in the Rich header. March 2018. https://securelist.com/the-devils-in-the-rich-header/84348/.

[6] Rascagnères, P.; Mercer, W. Who wasn't responsible for Olympic Destroyer. Virus Bulletin International Conference 2018, Montreal. https://www.virusbulletin.com/uploads/pdf/magazine/2018/VB2018-Rascagneres-Mercer.pdf.

[7] Kálnai, P.; Cherepanov, A. Lazarus KillDisks Central American casino. April 2018. https://www.welivesecurity.com/2018/04/03/lazarus-killdisk-central-american-casino/.

[8] Cherepanov, A. GREYENERGY: A successor to BlackEnergy. October 2018. https://www.welivesecurity.com/wp-content/uploads/2018/10/ESET_GreyEnergy.pdf.

[9] Cherepanov, A.; Lipovský, R. New Telebots backdoor linking Industroyer and notPetya. WeLiveSecurity.com. October 2018. https://www.welivesecurity.com/2018/10/11/new-telebots-backdoor-linking-industroyer-notpetya/.

[10] https://github.com/VirusTotal/yara/blob/master/libyara/modules/pe.c#L190.

[11] Webster, G.D.; Kolosnjaji, B.; von Pentz, Ch.; Kirsch, J.; Hanif, Z.D.; Zarras, A.; Eckert, C. Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage. In: Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 119--138, 2017. https://www.sec.in.tum.de/i20/publications/finding-the-needle-a-study-of-the-pe32-rich-header-and-respective-malware-triage.

[12] Cherepanov, A. Win32/Industroyer a new threat to industrial control systems. WeLiveSecurity.com. June 2017. https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf.

[13] Kafka, F. From Hacking Team to Hacked Team to…? Virus Bulletin International Conference 2018, Montreal. https://www.virusbulletin.com/uploads/pdf/magazine/2018/VB2018-Kafka.pdf.

[14] Soucek, J.; Tomanek, J.; Kálnai, P. Collecting Malicious Particles from Neutrino botnets. The Journal on Cybercrime & Digital Investigations, Botconf 2018 Conference proceedings, https://journal.cecyf.fr/ojs/index.php/cybin/article/view/22/24.

[15] ESET Research: Sathurbot: Distributed WordPress password attack. WeLiveSecurity.com. April 2017. https://www.welivesecurity.com/2017/04/06/sathurbot-distributed-wordpress-password-attack/.

[16] YARA PE module documentation. https://yara.readthedocs.io/en/v3.7.0/modules/pe.html.