Finding the Needle: A Study of the *PE32* Rich Header and Respective Malware Triage

George D. Webster[§], Bojan Kolosnjaji[§], Christian von Pentz[§], Julian Kirsch[§], Zachary D. Hanif, Apostolis Zarras[§], and Claudia Eckert[§]

[§] Technical University of Munich

Abstract. Performing triage of malicious samples is a critical step in security analysis and mitigation development. Unfortunately, the obfuscation and outright removal of information contained in samples makes this a monumentally challenging task. However, the widely used Portable Executable file format (*PE32*), a data structure used by the Windows OS to handle executable code, contains hidden information that can provide a security analyst with an upper hand. In this paper, we perform the first accurate assessment of the hidden *PE32* field known as the Rich Header and describe how to extract the data that it clandestinely contains. We study 964,816 malware samples and demonstrate how the information contained in the Rich Header can be leveraged to perform rapid triage across millions of samples, including packed and obfuscated binaries. We first show how to quickly identify post-modified and obfuscated binaries through anomalies in the header. Next, we exhibit the Rich Header's utility in triage by presenting a proof of concept similarity matching algorithm which is solely based on the contents of the Rich Header. With our algorithm we demonstrate how the contents of the Rich Header can be used to identify similar malware, different versions of malware, and when malware has been built under different build environment; revealing potentially distinct actors. Furthermore, we are able to perform these operations in near real-time, less than 6.73 ms on commodity hardware across our studied samples. In conclusion, we establish that this little-studied header in the *PE32* format is a valuable asset for security analysts and has a breadth of future potential.

1 Introduction

The shear volume of malware samples that analysts have to contend with makes thorough analysis and understanding of every sample impractical. As a result, effective and timely triaging techniques are vital for analysts to make sense of the collective information and focus their limited time on agglomerated tasks through uncovering commonalities and similar variants of malicious software. This in turn, allows analysis to better hone in their effort and avoid wasting costly cycles on previously analyzed or unrelated samples. Unfortunately, it is common practice for malware authors to design malware that hinders automated analysis and otherwise thwart triaging efforts; thereby allowing malware to operate under the radar for a longer period of time.

One of the common practices used in triaging samples is to leverage header information from the Portable Executable file format (*PE32*) [3, 6, 7]. This is primarily done as the derived information can: (*i*) reveal how the executable was built and who built it, (*ii*) provide an understanding of what the executable does, and (*iii*) identify entry points for disclosing packed and obfuscated code. For example, when investigating the *Rustock* Rootkit, the *PE32* Headers identified the location of the first deobfuscation routine [3]. Additionally, numerous clustering and similarity matching algorithms are often exclusively based on the data derived from the *PE32* file format [5, 11, 24, 26].

Unfortunately, malware authors are well aware of the valuable data contained in the *PE32* file format. As a result, they routinely take steps to strip or otherwise distort any useful information from the *PE32* format through packing binaries, adjusting compiler flags, and manually removing data in the header [1, 10]. While unpacking malware and performing manual reverse engineering can recover this useful information, the process is extremely costly. As stated by Yan et al. [27], "Who has the time to reverse all the bytecodes given that security researchers are already preoccupied with a large backlog of malware?" Needless to say, stripping the headers leaves little useful information available for triage.

Fortunately for security analysis, the *PE32* Header contains information that is often poorly understood or simply hidden. In this work, we perform an in-depth study of one of these hidden attributes commonly known as the Rich Header. While rich in information, this header is also common in malware, present in 71% in our random sample set, and is found in any *PE32* file assembled by the Microsoft Linker. Through the knowledge we derive from our in-depth study, we show how to properly dissect the information and explain what the resulting data means. We then study the extracted headers from malicious samples, which we gather from four distinct datasets. Leveraging this information, we present proof of concept methods to demonstrate the significant value the Rich Header can provide for triage.

Overall, in this paper, we show that the Rich Header field is valuable in triage and can be a catalyst for past and future work. As such, we provide the first accurate assessment of the Rich Header and detail how to extract its clandestine data. We then present a series of statistical studies and describe two proof of concept methods in which the Rich Header can be used in triage. The first method allows for the rapid detection of post-modified and packed binaries through the identification of anomalies. The next method leverages *machine learning* to perform rapid and effective triage based solely on the values in the Rich Header's *@comp.id* field; specifically the 516 unique *ProdID*, 29,460 distinct *ProdID* and *mCV* pairs, and their *Count* values we have identified across 964,816 malicious samples. This method can identify *similar malware variants* and *build environments* in 6.73 ms across 964,816 malware samples using only a consumer grade laptop. In summary, we prove that leveraging the data contained in this often forgotten and overlooked aspect of the *PE32* file format, called the Rich Header, establishes a major boon for performing analytic triage operations and opens the door for a plethora of future work.

In summary, we make the following main contributions:

- We present the first accurate and in-depth study of the Rich Header field and describe how to extract its data.
- We demonstrate how anomalies in the Rich Header can identify 84% of the known packed malware samples.
- We present a proof of concept approach that utilizes machine learning techniques to identify similar malware variants and build environments in near real-time, 6.73 ms, by only leveraging the Rich Header.

2 Background

In this section, we provide a background on the Portable Executable file format, commonly known as *PE32*.[1] We then present an overview of the compiler linking and describe how it creates the sections contained in the header.

2.1 Portable Executable File Format Headers

The Portable Executable file format was introduced by Microsoft to provide a common format for executable files across the Windows Operating System family [17]. As such, the format is the primary standard used for shared libraries, binaries, among other types of executable code or images in Windows. The Portable Executable file format is also often called the Common Object File Format (*COFF*), *PE/COFF*, and *PE32*.

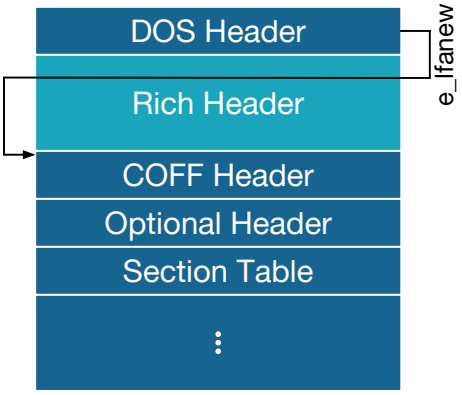

Fig. 1. High level view of the *PE32* format

The *PE32* format includes an MS-DOS stub for backwards compatibility, with the `e_lfanew` field pointing to the beginning of the COFF Header, as Figure 1 illustrates. The COFF Header, in turn, is followed by optional headers that control (among others) imports, exports, relocations, and segments [14]. Together, these headers contain valuable information for program execution, identification, and debugging. Including the base address of the image in virtual memory, the execution entry point, imported and exported symbols, and information on the code and data sections that form the program itself.

[1] For simplicity, we will use the 32 bit version of the Portable Executable file format. The 64 bit version behaves similarly.

The *PE32* Header is openly documented by Microsoft and as such its internal mechanics are well understood by the development community [13]. However, as we will discuss in this work, the *PE32* format does contain undocumented sections that have eluded understanding.

2.2 Compiler Linking

The typical process of building an executable image is subdivided into two parts: the compilation phase in which the compiler translates code written in a high-level language into machine code and the linking phase where the linker combines all produced object files into one executable image. These are both compartmentalized processes introducing a one-to-one relation between compile units (usually files containing source code) and the resulting object files.

The Microsoft Visual C++ (MSVC) Compiler Toolchain is a commonly used solution for building executable images from source code written in a variate of programming languages. During the compilation phase `cl.exe` provides the interface to the Front-End compilers (`c1.dll` and `c1xx.dd`) as well as the Back-End compiler `c2.dll`. Then, `c2.dll` will create object files (`.obj`) from intermediate files from the Front-End compilers. While creating the objects, the Microsoft Compiler assigns each object file an ID, which in this work referred to as the *@comp.id*, and stores the ID in the header of the respective object file. It is important to note that also the Microsoft assembler as well as the part of the tool chain that is responsible for converting resource files into a format suitable for linking generated *@comp.id*s.

Once the compilation phase is complete, `link.exe` will collect the objects needed and begin to stitch the *PE32* file or static library (`.lib`) together. Consequently, static libraries consisting of more than one object contain multiple *@comp.id*s. For executables and dynamic link libraries, `link.exe` builds up the Rich Header during generation of the appropriate *PE32* headers.

3 Rich Header

The Rich Header name originates from the fact that one of the elements in the header contains the ASCII values for "*Rich*." The header is an undocumented field in the *PE32* file format that exists between the MS-DOS and the COFF Headers. While rumors of its existence and speculation on its purpose have existed across multiple communities for a long time, it was not until July of 2004 that an article by *Lifewire* started to unveil information about the Rich Header [8]. Unfortunately, this article provided limited technically correct information and the drawn conclusions—especially regarding the purpose of the *@comp.id* field—were incorrect. Four years later, on January 2008, Trendy Stephen furthered the understanding of Rich Header by discovering some of the meaning behind the *@comp.id* field and a relatively correct assessment of how the checksum is generated [22]. Few months later, Daniel Pistelli released an article that provided a guide for extracting the Rich Header and portions of the *@comp.id* field [18].

Then, two years later, in November 2010, Daniel Pistelli updated his article with information describing how the high value bits in *@comp.id* correspond to a "Product Identifier" (referred to in this paper as *ProdID*) [18].

While the work of these pioneers provided crucial details on how to reverse engineer the Rich Header, aspects of the header are still poorly understood. Specifically, the full structure of the *@comp.id* has not yet been identified, information about how to map the *ProdID* is unknown, and mistakes were made regarding how to extract the fields. In this section we perform a technical deep dive of the Rich Header, which has not previously been completely and accurately described in any single source. We then explain how the header is added to *PE32* files, reveal the meaning behind the *ProdID*s, and present our algorithm for generating the hashes used to obfuscate the header.

3.1 Core Structure

During building, the Microsoft Linker (`link.exe`), via calling the function *CbBuildProdidBlock* [18], adds the Rich Header to the resulting binary. Although this action is performed during the building of the COFF Header, it is undocumented in the Microsoft specification [13] and begins before the official start of the COFF Header, as designated by the symbol `e_lfanew` in the MS-DOS stub. Additionally, this field is ubiquitous and cannot be disabled by compilation flags or by selecting different binary formats. The only notable exception is when the linker is not leveraged. For example, *.NET* executable files do not use the MSVC linker and these executables do not contain a detectable Rich Header.

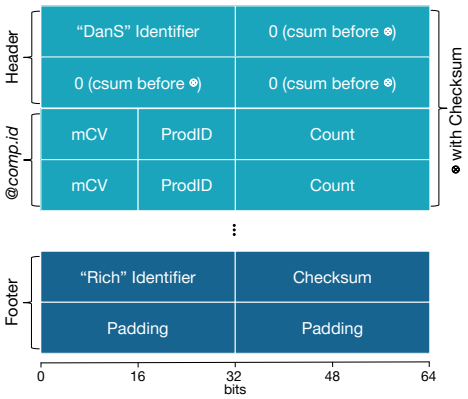

Fig. 2. Structure of the Rich Header

The Rich Header has been added as far back as 1998 with the release of Microsoft VC++ 6. Since then, each iteration of the Microsoft Toolchain adjusts how the header is generated and updates the *ProdID* mapping that the MSVC can generate. However, we suspect that this header has been included prior to VC++ 6. This is because we can have seen evidence of a potential Rich Header like field in samples that were generated before the release of VC++ 6. Unfortunately, it was not possible to confirm this belief because we were unable to obtain an older version of the Microsoft Linker and received only a smattering of samples generated before 1998.

Diving deeper, the generated structure of the Rich Header is composed of three distinct sections: the header, an array of *@comp.id* blocks, and the footer, as Figure 2 depicts. Together, these provide four core pieces of information:

(*i*) a checksum composed from a subset of the *PE32* MS-DOS header and the *@comp.id*s, (*ii*) the *ProdID* used when building the binary, (*iii*) the minor version information for the compiler used when building the product, and (*iv*) the number of times the linker leveraged the product during building.

The header of the Rich Header is composed of four blocks where each is `0x04` bytes in length. The first block contains the ASCII representation of "*DanS*"—it is speculated that "*DanS*" probably refers to *Daniel Spalding* who ran the linker team in 1998 [2]—while the next three blocks contain null padding. During linking, this section is XORed with a generated checksum value that is contained in the footer of the Rich Header.

The next section of the Rich Header is represented by an array of *@comp.id* blocks. Each block is `0x08` bytes in length and contains information related to the Product Identifier (*ProdID*), the minor version information for the compiler used to create the product (*mCV*), and the number of times the product was included during the linking process (*Count*). All fields are stored in little endian byte order and XORed with the previously mentioned checksum value. The *@comp.id* block consists of the following three values:

1. The *mCV* field contains the minor version information for the compiler used to build the *PE32* file. This version information allows the establishment of a direct relationship between a particular version of the Microsoft Toolchain and this *@comp.id* block in the Rich Header. For example, Microsoft's latest Visual Studio 2015 release ships version `14.00.23918` of the MSVC compiler (`cl.exe`). Therefore, object files created by this compiler will contain the value of `0x5d6e`. During the linking process for the building of a *PE32*, the value will be added into the produced *PE32*'s Rich Header in the *mCV* field of the *@comp.id* block representing this object.
2. The *ProdID* provides information about the identity or type of the objects used to build the *PE32*. With respect to type, each Visual Studio Version produces a distinct range of values for this field. These values indicate whether the referenced object was a C/C++ file, an assembly blob, or a resource file before compilation as well as a subset of the compilation flags. For example, a C file compiled with Visual Studio 2015 will result in the value `0x104` being copied into the Rich Header as *ProdID* in all *PE32* files that include the respective object file.
3. The *Count* field indicates how often the object identified by the former two fields is referenced by this *PE32* file. Using a simple C program as an example, this fields will hold the value `0x1` zero-extended to span 32 bits, indicating that the object file is used once by the *PE32*.

The final section of the Rich Header, the footer, is composed of three blocks of information. The first block is `0x04` bytes in length and represents the ASCII equivalent of "*Rich*". The next `0x04` bytes are the checksum value that are used as the XOR key for enciphering the Rich Header. The final block section is used as padding, typically null, and ensures that the total length of the Rich Header is a multiple of 8. Unlike the previous two sections, the footer is not XORed with the checksum value.

3.2 Hashes Contained Within the Rich Header

In the Rich Header, the checksum value appears at four distinct places, as shown in Figure 2. The first three occurrences are located immediately after the ASCII equivalent of "DanS". As the linker initially places null values in this location, they only appear after the header is XORed with the checksum value. The final checksum is located in the footer and immediately after the ASCII equivalent of "Rich".

While checksum values are traditionally straight forward to generate, the Rich Header's checksum has interesting properties. Specifically, only 37 of each *@comp.id*'s 64 bits are calculated. As such, we present the following algorithm [2], based on our reverse engineering work, which produces a valid Rich Header checksum.

The Rich Header checksum is composed of two distinct values c_d and c_r that are summed together. To calculate c_r, we define the rol operator, which zero extends its first argument to 32 bits and then performs a rotate left operation equal to the second arguments value of the first argument's bits. We define rol as:

$$\begin{aligned}\mathrm{rol}(val, num) := ((val \texttt{ << } num) \texttt{ \& 0xffffffff}) \mid \\ \mid (val \texttt{ >> } (32 - num))\end{aligned}$$

where `<<` and `>>` denote logical left and right shift, and `|` and `&` are the binary OR/AND operators. Then, the distinct parts of the checksum *csum* are calculated in the following way:

1. For c_d, all bytes contained in the MS-DOS header with the "*e_lfanew*" field (offset `0x3c`) set to `0` are rotated to the left by their position relative to the beginning of the MS-DOS header and summed together. Zeroing the "*e_lfanew*" field is required as the linker can not fill in this value because it does not know the final size of the Rich Header. Therefore is unable to calculate the offset to the next header. Let n denote the length of the MS-DOS header in bytes (most commonly `0x80`) and let DOS_i be the i-th byte of the (modified) MS-DOS header:

$$c_d = \sum_{i=0}^{n} \mathrm{rol}(\mathrm{DOS}_i, i)$$

2. To calclulate c_r, the algorithm first retrieves the list of m *@comp.id* blocks. Then the algorithm combines the corresponding *mCV* and *ProdID* parts into one 32 bit value. Finally, this value is rotated to the left by its respective *Count* value:

$$c_r = \sum_{j=0}^{m} \mathrm{rol}(ProdID_j \texttt{ << } 16 \mid mCV_j, Count_j \texttt{ \& 0x1f})$$

[2] For a copy of the checksum algorithm, please see Section 8.

It is noteworthy that despite the fact that *Count* beings as a 32 bit field, the checksum algorithm only considers the least significant byte value (& `0xff`). Combined with the fact that $m \equiv n \mod 32 \implies \text{rol}(v, n) = \text{rol}(v, m)$, it is sufficient to perform the calculation as indicated above.

The two values c_d and c_r, and the size of the MS-DOS header (`0x80`) are then added together to form the final checksum value:

$$\text{csum} = \texttt{0x80} + c_d + c_r$$

3.3 Generation of *@comp.id* and *ProdID*

The *@comp.id* is generated for each object file before linking. The type of the object being created is determined during the creation of the object file. With this information, the respective generator (see Table 1) will then assign a *ProdID* and *mCV* that map to object type and the Visual Studio release version in which the object was compiled.[3] For instance, a *ProdID* value of 255 to 261 corresponds to a Visual Studio 2015 Resource, Export, Import, Linker, Assembler, C, and a C++ file respectively. The same range of values can be shifted to base values `0xab`, `0xcf`, and `0xe1` which correspond to Visual Studio 2010, 2012, and 2013. Additionally, the *ProdID* is adjusted based on the compilation flags used to create the object. To date we have identified that the MSVC Toolchain is capable of assigning 265 *ProdID*. During our research we found that the generated *ProdID*s cannot be manually changed without patching the compiler backend.

ProdID	**VS Release**	**Object Type**	**Generator**
`0x105`	2015	C++	`c2.dll` via `cl.exe`
`0x104`	2015	C	`c2.dll` via `cl.exe`
`0x103`	2015	Assembly	`c2.dll` via `ml.exe`
`0x102`	2015	Linker	`link.exe`
`0x101`	2015	Imported sym.	`c2.dll` via `cl.exe`
`0x100`	2015	Exported sym.	`c2.dll` via `cl.exe`
`0xff`	2015	Resource file	`cvtres.exe`

Table 1. Subset of *ProdID*s generated by Visual Studio 2015

In cases where a *ProdID* is already present, such as a third party static library (`.lib`) containing multiple object files, the linker uses the preexisting *ProdID*s and *mCV*s. Inside of the library, the data is represented as a linked list. Interestingly enough, in our research we have found that these *ProdID*s do not necessarily correlate to what the MSVC Toolchain can generate. Specifically, we have identified 251 *ProdID*s that cannot be generated by MSVC and appear to map to either a bundled library or the libraries supplied by major corporations.

3.4 Adding the Rich Header to the *PE32* File Format

During the build process, the section that generates the data contained in the Rich Header is located in the Microsoft Compiler backend (c2.dll). The Microsoft

[3] For a mapping of *ProdID*s that the MSVC Toolchain can generate, see Section 8.

Linker (link.exe) then collects the data required to build the Rich Header and places it in the generated *PE32* file.

4 Statistical Analysis

To study the effectiveness of using the Rich Header for triage operations, we developed a custom extractor and studied the values extracted across approximately one million malware samples.

4.1 Data Sources

We leveraged four sets of *PE32* samples during our evaluation. The first set is composed of 964,816 randomly selected malicious *PE32* samples from 2015 and is supplied by VirusShare [19]. The second set contains 1875 samples from the Mediyes dropper [28]. The third set contains 2031 samples related to the Zeus derivative Citadel [21]. The final set is composed of 293 samples associated to the APT1 espionage group [4].

In total, these binaries represent a diverse set of malware types that range from traditional criminal malware, highly advanced state-sponsored malware, and programs which have not been confirmed to be malicious but are highly suspicious. It is worth to mention that in order to study the effectiveness of the Rich Header during triage, we made no efforts towards unpacking or deobfuscating these samples.

4.2 Extracting the Rich Header

While the Rich Header has been identified for a number of years, to our knowledge, no articles have completely and accurately explained the header's structure. As a result, we found that most common triage engines and libraries that parse the *PE32* Headers either ignore, do not fully process, or perform incorrect parsing of the Rich Header. For example, two of the most common and openly available malware triage systems, Viper and MITRE's CRITs, do not properly extract this field. In the case of Viper, the supplied *PE32* Header extractor ignores the Rich Header field entirely, whereas CRITs will attempt to process the Rich Header but performs an incomplete extraction. Specifically, CRITs will only process the first `0x80` bytes of the Rich Header and does not extract the fields contained in the *@comp.id* data structure. Unfortunately, this is not unique to triage systems. When looking at major *PE32* parsing libraries, we found that the very popular PEFile has a similar issue to CRITs in that it also only parses the first `0x80` bytes of the Rich Header and also does not extract the values contained in the *@comp.id*. Furthermore, the *PE32* extraction script, `pescanner.py`, from the Malware Analyst's Cookbook [9] ignored the Rich Header field entirely. Last, when the Rich Header is attempted to be parsed, we have found it to be common to only attempt to identify the Rich Header at location `0x80`. As a result, we developed a custom service to accurately extract the Rich Header information.

4.3 Information Gathering

To generate our information we used an automated infrastructure based on SKALD that executed five services across our datasets [25]. These services (*i*) extracted the Rich Header using our custom tool, (*ii*) performed Yara signature matching with 12,693 signatures provided by YaraExchange [15], (*iii*) retrieved malicious scan results from VirusTotal, (*iv*) performed identification on the compiler and any potential packer used to create the sample, and (*v*) generated pseudocode via IDA Pro.

4.4 Statistical Results

With our gathered information we performed a series of statistical studies. This was done to better understand the Rich Header's prevalence in malicious samples and identify which packers or compilers omit the Rich Header. Additionally, we developed a statistical check that is capable of rapidly identifying packed and post-modified *PE32* files, leveraging data only contained within the Rich Header.

Samples with a Rich Header. We identified that a surprisingly high percentage of samples contain the Rich Header, as shown in Table 2. For instance 71% of the random sample set and 98% of APT1 sample set contained parseable versions of the Rich Header. This is surprising as our initial assumption was that malware authors would use a variety of compilers when creating samples and potentially attempt to strip the Rich Header. However, the results show that the majority of malware authors are in fact leveraging the Microsoft Linker and pay no mind to the Rich Header.

Family	Total	Rich Header	Percent
Random Set	964,816	683,238	71%
APT1	292	286	98%
Zeus-Citadel	1928	717	37%
Mediyes	1873	30	2%

Table 2. Samples containing a Rich Header with total percentages rounded

Based on the above information, we conclude that the Rich Header is commonly found in malware and that malware authors do not deliberately strip the Rich Header. Furthermore, we can conclude that compilation of malicious binaries are most often done using compilers that leverage the Microsoft Linker.

Compilers and Packers without a Rich Header. While the high rate for malware containing a Rich Header is positive for triage, this was not a uniform result. Specifically, some malware variants reported a low match for samples containing the Rich Header, such as Mediyes reporting 2%. In Section 2.2 we discussed that the Rich Header is generated by the Microsoft Linker. This implies that compilation tools not using the official Microsoft Linker should not generate the Rich Header. While this can explain why some samples do not include the Rich Header, in this Section we further explore other reasons behind the absence of the field. Specifically, we identify common tools and packers used by malware to either strip or corrupt the Rich Header.

To do this we used our service that performs compiler and packer identification to scan all samples without a Rich Header. This was done to identify if there are any commonalities with these sample. As Table 3 shows, the percentage of samples built by either Borland C++ Builder or MinGW, which is based on GCC, is relatively high and accounts for approximately a third of all samples that do not contain a Rich Header in the random and Zeus-Citadel datasets. However, this was not the case in the APT1 and Mediyes dataset. Upon further analysis, we identified that most packers, while sometimes introducing anomalies, did not often strip the Rich Header from samples. With respect to the Mediyes set, we had a high rate of matches for the Themida Packer [23]. As we discuss further in Section 4.4, Themida is one of the packers that rewrites the entire *PE32* file and does not include the Rich Header. Instead, we identified that the absence of a Rich Header was a result of corruption caused during the packing of the sample.

Family	No Rich	dUP	MinGW	Borland
Random Set	157,497	135,115	25,387	63,283
APT1	6	4	1	0
Zeus-Citadel	1211	673	93	398
Mediyes	1843	1787	0	194

Table 3. Samples not containing a Rich Header

To identify if other packers caused similar corruption, we leveraged our identification service again to detect the most common packers used by our malware datasets. Our results showed that UPX, ASPacker, mingw, dUP, and the Nullsoft Scriptable Install System were the top five most commonly used packers. As we already understood that samples created with mingw and dUP will remove or otherwise corrupt the Rich Header, we manually created test samples with variants of UPX (v1, v2, and v3.91), ASPacker, and Nullsoft. In every manual test case, we were unable to cause a corruption or exclusion of the Rich Header field.

Identifying Modified Binaries Based on Rich Header Corruption. In our previous results, we found that it was uncommon for malware authors to deliberately strip the Rich Header. As such, we re-evaluated our samples to search for cases where the Rich Header was inadvertently corrupted.

The first approach we took was to identify cases where the Rich Header contained duplicate *@comp.id* blocks. We took this approach because under normal operation, the Microsoft Linker should never produce duplicate entries. This is because during the linking process, the Microsoft Linker will search for existing instances of the *ProdID* and *mCV* and if identified, will increment the number of times it was used, *Count*, to the existing entry.

The second approach we took was to re-calculate the Rich Header checksum and compare it to the sample's reported Rich Header checksum. This was done as an unsuccessful check would indicate that either the MS-DOS Header or the Rich Header was modified after the linking process; potentially revealing Trojanized or post modified binaries.

As Table 4 shows, the amount of malicious samples containing a corrupted Rich Header varies and can rise upwards to 50% based on the malware family.

Additionally, across the random one million dataset, this corruption occurred approximately 31% of the time. Knowing this and the fact that no official Microsoft Linker should produce these forms of corruption, identifying corruption of the Rich Header can be a fast and efficient triage step to use for screening samples for potential maliciousness.

@comp.id and mCV Values Present in Malware. To develop an understanding of how we can potentially leverage the Rich Header for more advanced triage operations, we studied the *@comp.id* values in our malware datasets. By doing so, we identified 516 unique *ProdID*s. This was surprising as all versions of the MSVC Toolchain, dating back to VS++ 6, are only capable of generating 265 *ProdID*s. While researching the 251 unknown *ProdID*s, we identified that these appear to more than likely correlate to bundled libraries and major corporations. However, while in practice this assumption appears to be accurate, we cannot conclusively confirm this.

Family	Total	Dup. ID	csum Err
Random Set	683,238	15,006	137,965
APT1	286	0	34
Zeus-Citadel	717	17	357
Mediyes	30	0	0

Table 4. Samples containing a Rich Header that have duplicate entries and invalid checksums

Digging in deeper, we discussed in Section 3 that the *ProdID* is paired with the *mCV*. Thus, potentially providing more fine grained information for identifying specific objects. To confirm this we created tuples of all the *ProdID* and *mCV* pairings. We then single out 29,460 distinct *ProdID* and *mCV* pairs across our approximately one million malware samples. These numbers show relatively substantial variability in the *@comp.id*s found in malware and malware authors build environments.

5 Case Study

The data obtained in Section 4 showed promise in using the Rich Header for more complex triage operations. This is especially true considering the vast majority of our datasets are from the same date range and the fact that the Rich Headers of malicious samples contain numerous *@comp.id*s along with the number of times the object was used during linking.

In order to demonstrate the potential of leveraging the Rich Header in future work, we created a basic proof of concept machine learning algorithm. The algorithm was designed to only process the *@comp.id* values contained in the Rich Header. Specifically, the values for *ProdID*, *Count*, and *mCV*. As the Rich Header identifies linked object and version information of the build environment, our algorithm is specifically focused on identifying similar samples, based on linked objects, and also samples using a similar build environment. In crafting the algorithm, we used a feature hashing strategy which transformed the features into a *50-dimensional* vector. We then leveraged a *Stacked Autoencoder* to turn

our data into a denser, lower-dimensional space. Finally, in order to improve performance and allow us to scale to support datasets containing millions of malware samples, we utilized a *Ball Tree* for fast storage and retrieval of the vectors.

In the following case studies, we demonstrate the ability of solely using the Rich Header to perform similarity matching leveraging our proof of concept machine learning algorithm. In the case studies, we compare the exemplar samples, selected at random, with the collected vector similarities from the *Ball Tree* populated by the random one million, APT1, and Zeus-Citadel datasets, and analyze their closest matches. For our ground truth in the case studies, we compare the results of our algorithm to the results returned by Kaspersky and Symantec Antivirus, as implemented by VirusTotal, and perform manual reverse engineering. We selected this ground truth method primarily due to the limited matches across Yara and the high percentage of no detection or generic signatures across the other popular AV vendors.

5.1 Similarity Matching with the APT1 Dataset

We selected three exemplar samples from the APT1 dataset for our first case study. APT1 was selected as the actor is a relatively skilled APT and 98% of the samples in the APT1 dataset contain a Rich Header.

We randomly selected our first exemplar sample, E1. Kaspersky classifies this sample as *HEUR:Trojan.Win32.Generic* which means that through heuristic analysis, Kaspersky believes that this is a Trojan but has not classified the sample further. When querying our algorithm, we identified that it had an identical Rich Header feature vector with another APT1 sampled, which we will refer to as E1-R1. Inspecting E1-R1, Kaspersky classified the sample also as *HEUR:Trojan.Win32.Generic*. While a generic classification does not tell us much, manual analysis of the generated source code, produced by IDA Pro, confirmed that these two samples were in fact identical.

Going a step further, we then queried the nearest neighbor to E1. This returned three samples: E1-N-R1, E1-N-R2, and E1-N-R3. All three matches were also contained in the APT1 dataset and shared the *HEUR:Trojan.Win32.Generic* Kaspersky classification. Our algorithm reported that the distance between these vectors was `1`; the smallest possible difference without the vectors being identical. We then performed manual analysis and identified that the generated source code produced by IDA Pro for E1-N-R1 was identical to our exemplar. However, as the vectors were slightly off, we further analyzed the cause for this and concluded that the variance was caused by slightly different build environments when compiling the binaries.

The other two nearest neighbor matches, E1-N-R2 and E1-N-R3, produced even more interesting results. In both cases, the generated source code produced by IDA Pro had slight differences. In the case of E1-N-R3, E1-N-R3 adds a call to function $FlushFileBuffers$ right after it writes the buffer to a file. Furthermore, E1-N-R2 seemed to build upon the changes made to E1-N-R3. Specifically, E1-N-R2 includes an additional change in that E1-N-R2 adjusted how it wrote the

buffer to files. In our exemplar sample, E1 first writes the buffer to the file and then performs a second write that adds `\r\n` to the file. In the case of E1-N-R2, the sample does not write `\r\n` to the file and instead calls `strcat` on the buffer in order to add `\n` to the buffer before it writes the buffer to the file.

Our second exemplar, E2, was selected from the APT1 dataset because its signature was different than E1 and it shares its feature vector with no other samples. After running our algorithm, we identified E2-N-R1 as the nearest neighbor to E2 at a distance of 1.732. While it is reasonable to argue that the distance is very near, it is indicative of a clear similarity between the samples.

When analyzing the results, both E2 and E2-N-R1 are classified by Kaspersky as "Agents". However, the generated source code produced by IDA Pro for both samples is quite different and the programs have different functionality. To understand why our algorithm identified this as a match, we performed additional research on the binaries and found that both E2 and E2-N-R1 are very small, had a nearly identical import table with only one variation, and were packed with Armadillo v1.71. Looking at the Rich Header vectors we found that the vast majority of the objects imported all had identical version information; which led us to conclude that the samples were more than likely built on the same machine or the machines at least had an identical build environment. Open-source research further validated this opinion as both samples were used by APT1 in cyber operations [20]. While not a direct match in terms of functionality, this example demonstrates the power in using the Rich Header to identify not only similarly behaving malware but also malware that is related because the malware is presumably built on the same machine.

Our final exemplar, E3, was selected as it had five samples that shared the same Rich Header feature vector: E3-R1, E3-R2, E3-R3, E3-R4, and E3-R5. In all cases, these samples ended up being members of the APT1 dataset and shared the *HEUR:Trojan.Win32.Generic* Kaspersky classification. Manual reverse engineering also showed that the samples shared a nearly identical code base and performed the same functionality.

We then queried our algorithm for the nearest neighbors to E3 that were not in the APT1 sample set. The query returned six samples at a distance of 2.236: E3-N-R1, E3-N-R2, E3-N-R3, E3-N-R4, E3-N-R5, and E3-N-R6. Kaspersky classified E3-N-R2, E3-N-R4, E3-N-R5, and E3-N-R6 as *HEUR:Trojan.Win32.Generic*. E3-N-R1 and E3-N-R3 were classified by Kaspersky at *Net-Worm.Win32.Cynic.in* and *Net-Worm.Win32.Cynic.am*, respectively. However, although the Kaspersky classifications were different, manual analysis revealed that all E3-N-R* samples were nearly identical to themselves. The only differences between the samples in this cluster were caused by artifacts left by the obfuscation engines and by the language settings on the build environment. Furthermore, the two clusters for the same vector, E3 and E3-R*, and nearest neighbors, E3-N-R*, were remarkably similar in functionality.

During the evaluation we identified that the similarity matching algorithm produced very strong results for the exemplar samples E1 and E3. However, with

E2 and E3, the algorithm further identified samples of similar nature and with a similar build environment.

5.2 Similarity Matching with the Zeus-Citadel Dataset

In this case study we opted to explore the results of two exemplar samples from the Zeus-Citadel botnet. We selected this dataset because the Zeus-Citadel actors are typical of basic cyber criminals and as such have a different target and mission than the actors behind the first test case.

Kaspersky classified our first exemplar, E4, as a generic Trojan. Our algorithm though was able to identify 23 similar samples, E4-R*, that shared the feature vector of E4. Kaspersky classified them as either *HEUR:Trojan.Win32.Generic* or *not-a-virus:AdWare.Win32.FakeDownloader.ac* whereas Symantec identifies all the E4-R* samples as *Trojan.Gen* or *Trojan.Zbot*. When comparing the IDA Pro generated source code, we confirmed that the E4 and E4-R* samples were nearly identical; the differences in E4 and E4-R1 are that sections of the code has moved under different functions and that E4 uses a "for loop" while R4-R1 uses a "do while" for their XOR algorithm. Thus, the difference in E4 and E4-R* appear to be related to slightly different version, compiler optimization, or artifacts left by the obfuscation engines.

When looking at the nearest neighbor cluster for E4 we identified four additional samples: E4-N-R1, E4-N-R2, E4-N-R3, E4-N-R4. While Kaspersky classified the sample as *HEUR:Trojan.Win32.Generic*, Symantec identified E4-N-R1 and E4-N-R3 as clean. However, when looking at the samples, we observed only a slight variation in that E4-N-R* ran the XOR loop 220,712 times where E4 and E4-R* ran the XOR loop 51,700 times. As with E4-R*, E4-N-R* also moved code segments into different functions. When verifying the Rich Header we observed that the reason for being classified as a nearest neighbor was because of variations in the number of times one product was included. This is a clear example where using Rich Header values as a triage system could prove useful for an investigative team by identify similar malware samples from potentially different version.

The next exemplar, E5, shares its vector hash with 36,606 samples. This is notably high, no less so due to the fact that Kaspersky fails to identify 16,123 of those samples with a classification of any kind, not even the most generic of names. However, when comparing the IDA Pro generated source code, we observed only small variations; specifically the value for a constant was changed.

The nearest neighbor grouping for, E5, has a distance of 2 and contained a total of 1,567 samples, where 511 samples have no Kaspersky listing. In fact, the majority of samples in both groups are listed as generic Trojans, *HEUR:Trojan.Win32.Generic*. While the inclusion of a known Zeus-Citadel sample is not enough to convict these samples as Zeus-Citadel members, it provides an interesting jumping off point for analysis.

5.3 Similarity Matching with the Mediyes Dataset

In our final case study, we selected a random Mediyes sample, E6, to use as our exemplar. We chose this sample because it would allow us to perform an out-of-set comparisons as the Mediyes dataset was not originally vectorized and included in the *Ball Tree*.

Querying our algorithm for identical Rich Header feature vectors, we received a list of 266 other samples: E6-R*. When querying for the nearest neighbors we receive 86 additional samples, E6-N-R*, with a distance of `1`. Analysis of the IDA Pro generated source code showed a strong correlation between the samples. Furthermore, the vast majority of both E6-N-R* and E6-R* were classified by Kaspersy as Zango samples; an instance of adware frequently associated with Mediyes.

6 Future Work and Limitations

In this paper we present an important yet little-studied header of *PE32*, the Rich Header. We show that the Rich Header has been largely ignored by malware authors and is not removed by most packers and obfuscation engines. In fact, 71% of the 964,816 samples in our random dataset include the Rich Header. Our experiments revealed that the Rich Header contains useful information that can be leveraged by defenders; analysis of the data contained within the header allows for the rapid triage of samples using a cost effective approach. This is true even for samples that are stripped and contain little to no *PE32* Header information.

We strongly believe that by leveraging the Rich Header, current and future triage algorithms will perform a more accurate and cost effective triage functionality. In future work, we will explore how to combine the Rich Header features with other aspects of the *PE32* file format to generate robust similarly matching and clustering algorithms. As the Rich Header artifacts helps to identify similar malware as well as characterizing the build environment in which the malware was built, this presents new opportunities for attribution and tool-chain identification.

Furthermore, as knowledge of the Rich Header grows, it is understandable that malware authors will attempt to obfuscate this field. This is an expected outcome but also presents interesting future work potentials when the Rich Header is combined with additional features. This is because leveraging compiler fingerprinting and additional *PE32* header information can be used to determine if the Rich Header should be included and approximate the expectant values of this field. As such, future algorithms can identify anomalies in the Rich Header field. This increases the complexity required when performing obfuscation and adds resiliency.

7 Related Work

Leveraging the data derived from the *PE32* file format has been widely explored for triage purposes. One common technique, as shown by Mandiant's Imphash,

is to generate a hash of the values located in the *PE32* Import Address Table (IAT) [11]. These hashes are then used in analytic queries and by machine learning algorithms to identify similar strains and families of malware. In this vein, JPCERT recently released Impfuzzy to improve upon this technique through incorporation of a fuzzy hash [24]. However, these algorithms require accurate IAT and their accuracy is greatly reduced if the malware strips or otherwise provides a misleading IAT.

In light of this issue, more advanced techniques use additional meta data that can be derived from the *PE32* file format. For example, PEHash uses the structural characteristics of the *PE32* file format to generate hashes that are then used in clustering operations [26]. Unfortunately, the above methods only work well when the data is available and not being misconstrued.

Identifying the weaknesses in this approaches, specifically PEHash's lack of robustness, Jacob et al. [5] expand upon these method by focusing their efforts on the *PE32*'s code section. While this approach is more tamper resistant, it is still not immune. On the other side of the spectrum, Perdisci et al. [16] focused their efforts on using patter recognition to identify packed samples and then send those samples to universal unpacking algorithms before matching occurs. However, while this process does reduce the cost and improve the accuracy in clustering and similarity matching, unpackers are known to be unreliable and exceedingly expensive [12, 16].

In a change of pace, our work illustrates a hidden aspect of the *PE32* file format, the Rich Header, that has been largely ignored by malware authors. Using this section of the *PE32* file formation, we show how to cheaply identify packed malware, perform similarity matching solely using this field, and identify malware that was created using similar build environments. Our work does not aim to directly compete with the existing research. Instead the knowledge gained from our novel approach aims to be a catalyst for triage when combined with these, and other, triage techniques. In turn, this work enables existing and future algorithms to provide better results, be more resistant to tampering due to the wider scope, and improve returned information by allowing matching based not just on the samples characteristics but also the characteristics of the build environment used to create the samples.

8 Conclusion

In this paper we performed an in-depth study of the Rich Header and showed its significant potential in being leveraged for triaging malicious samples. To the best of our knowledge, this assessment of the Rich Header is the most complete and accurate report for this hidden and undisclosed section of the *PE32* header so far. With this knowledge, we created a custom Rich Header parser and extracted the headers' contents in over 964,816 malicious samples. We demonstrated the Rich Header's potential in enabling the rapid triage of malicious samples. By doing so, we showed how to leverage the Rich Header to identify post-modified and packed *PE32* files, detecting 84% of the known packed malware samples.

We also demonstrate the value in leveraging the Rich Header by developing a proof of concept machine learning algorithm and performing three case studies. In these studies we are capable of rapidly returning results in 6.73 ms using a single CPU core, identifying similar malware variants, and highlight malware developed under the same build environments.

Availability

The feature extraction methods and additional reference material have been open-sourced under the Apache2 license. They can be freely downloaded at the following location:

`https://holmesprocessing.github.io`

Acknowledgments

We thank our shepherd Pavel Laskov and the reviewers for their valuable feedback. We are thankful to the Technical University of Munich for providing ample infrastructure to support our development efforts. Additionally, we thank the the German Federal Ministry of Education and Research under grant 16KIS0327 (IUNO) and the Bavarian State Ministry of Education, Science and the Arts as part of the FORSEC research association for providing funding for our infrastructure. We would also like to thank the United States Air Force for sponsoring George Webster in his academic pursuit. Lastly, we would like to thank Microsoft Digital Crimes Unit, VirusTotal, and Yara Exchange for their support and valuable discussions.

References

1. H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the Analysis of the Zeus Botnet Crimeware Toolkit. In *Annual International Conference on Privacy Security and Trust (PST)*, 2010.
2. R. Cafe. Microsoft's Rich Signature (Undocumented) - Comments. `http://rcecafe.net/?p=27`, February 2008.
3. K. Chiang and L. Lloyd. A Case Study of the Rustock Rootkit and Spam Bot. In *The First Workshop in Understanding Botnets*, 2007.
4. M. Intelligence. APT1: Exposing One of China's Cyber Espionage Units. *Mandian.com*, 2013.
5. G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A Static, Packer-Agnostic Filter to Detect Similar Malware Samples. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2012.
6. K. Kendall and C. McMillan. Practical Malware Analysis. In *Black Hat Conference, USA*, 2007.
7. B. Kolosnjaji, A. Zarras, T. Lengyel, G. Webster, and C. Eckert. Adaptive semantics-aware malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 419–439. Springer, 2016.

8. Lifewire. Things They Didn't Tell You About MS Link and the PE Header. (29A), 2004.
9. M. Ligh, S. Adair, B. Hartstein, and M. Richard. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. Wiley Publishing, 2010.
10. R. Lyda and J. Hamrock. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security & Privacy*, (2):40–45, 2007.
11. Mandiant. Tracking Malware With Import Hashing. `https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html`, January 2014.
12. L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, Generic, and Safe Unpacking of Malware. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.
13. Microsoft. Microsoft Portable Executable and Common Object File Format Specification, Rev. 8.3, 2013.
14. Microsoft. Common Object File Format - KB121460. `https://support.microsoft.com/en-us/kb/121460`, 2016.
15. M. Parkour and A. DiMino. Deepend Research. `http://www.deependresearch.org/2012/08/yara-signature-exchange-google-group.htm`, May 2015.
16. R. Perdisci, A. Lanzi, and W. Lee. Classification of Packed Executables for Accurate Computer Virus Detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.
17. M. Pietrek. An In-Depth Look Into the Win32 Portable Executable File Format. *MSDN magazine*, 17(2):80–90, 2002.
18. D. Pistelli. Microsoft's Rich Signature (Undocumented), 2012.
19. J.-M. Roberts. Virus Share. `https://virusshare.com/`, Apr 2016.
20. S. Sarméjeanne. The HTran Tool Used to Hack Into French Companies. `https://www.lexsi.com/securityhub/the-htran-tool-used-to-hack-into-french-companies/?lang=en`, Aug 2011.
21. R. Sherstobitoff. Inside the World of the Citadel Trojan. *Emergence*, 9, 2012.
22. T. Stephen. Rich Header. `http://trendystephen.blogspot.de/2008/01/rich-header.html`, Jan 2008.
23. O. Technologies. Themida - Advanced Windows Software Protection System. `http://www.oreans.com/themida.php`, Jan 2016.
24. S. Tomonaga. Classifying Malware Using Import API and Fuzzy Hashing -Impfuzzy-. `http://blog.jpcert.or.jp/2016/05/classifying-mal-a988.html`, May 2016.
25. G. D. Webster, Z. D. Hanif, A. L. Ludwig, T. K. Lengyel, A. Zarras, and C. Eckert. SKALD: a scalable architecture for feature extraction, multi-user analysis, and real-time information sharing. In *International Conference on Information Security*, pages 231–249. Springer, 2016.
26. G. Wicherski. peHash: A Novel Approach to Fast Malware Clustering. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.
27. W. Yan, Z. Zhang, and N. Ansari. Revealing Packed Malware. *IEEE Security & Privacy*, 6(5):65–69, 2008.
28. V. Zakorzhevsky. Mediyes - The Dropper With a Valid Signature. `https://securelist.com/blog/research/32397/mediyes-the-dropper-with-a-valid-signature-8/`, March 2012.