"PIMP MY PE": PARSING MALICIOUS AND MALFORMED EXECUTABLES

Nick Hnatiw
Tom Robinson
Casey Sheehan
Nick Suan

Sunbelt Software, 33 N. Garden Avenue
Clearwater FL 37855, USA 727-562-0101
{nickh,tomr,caseys,nicks}@sunbelt-software.com

Abstract

A foundational requirement in the security world is the capability to robustly parse and analyze Windows Portable Executable files. Coping with the full spectrum of PEs found in the wild is, in fact, quite challenging. While white files are typically well structured, malicious files can be quite difficult to analyze, often due to deliberate malformations intended to stymie static analysis. In this paper we will survey and attempt to classify some common and interesting malformations we have studied in our work at Sunbelt Software. We will analyze PE structural information, discuss the PE specification, and highlight specific hurdles we have overcome in the course of developing a parsing facility capable of dealing reliably with the full range of images found in the wild, especially malware. We will also cover specific problems we faced along the way, examine structural heuristics we've developed in the course of classifying common malformations, and include a discussion of some interesting tools and techniques we've developed.

1 Introduction

The four of us work at Sunbelt Software in the Core Anti-Malware Technology group, which is responsible for the VIPRE malware detection and remediation engine, a key component of Sunbelt's family of consumer and enterprise security products.

In this paper we delve deeply into the inner details of the Portable Executable format, and assume the reader has some understanding of the relevant PE file structures. For a gentler introduction to this material, consult the references, especially Matt Pietrek's seminal work [1].

1.1 The Need

At Sunbelt, we need the ability to parse all executables present on our customers' machines. The goal for our framework is to parse any PE into a robust internal representation (IR) for subsequent analysis and possible remediation.

1.2 The Problem

Building a flexible and robust parsing facility is not a trivial task. While there is ample material in the literature providing a basic foundation (anyone remember PEDUMP?), these solutions generally perform poorly on PEs in the wild. Many wild images are packed or otherwise post-processed, frequently in an attempt to disrupt static analysis and hide malicious content. Thus, there are a large number of what we might call "pimped" executables in the wild that are challenging to analyze, identify and neutralize.

Many technologies are involved in identification and the remediation of these threats, but fundamentally, these techniques are only as good as the PE parser. If the parser fails to parse a potential threat, we deem it a "malformed executable", our threat engine returns a "no threat" result to the application, the threat is white-listed, and might subsequently be allowed to execute.

Our specific problem is that we need to mimic the behavior of the Windows loader as closely as possible. If Windows loads an image, however malformed, our parser should likewise be able to parse it into a valid IR; by the same token if Windows won't load it, we shouldn't parse it either (it is safe to consider it malformed).

1.3 The Solution

The fundamental goal of our project is to create a parsing facility capable of processing all PE threats that have potential to run on a Windows platform. Our general approach has been to compare performance of our parser against a large dataset of both known good and known malicious PEs, and verify all OS-loadable images can be correctly parsed. In developing our solution, however, we were surprised by the high frequency with which we encountered malformations when scanning our blacklists; so many, in fact, that we were forced to adopt an incremental approach to refining the function of our parser. In other words, we developed a regression where we could iteratively scan a file repository with our parser, catalog its results, and verify the parser performance didn't regress as we "improved" it.

A key part of our effort has been to identify and track common image malformations, which we classify via a mechanism we call an *anomaly.* The analysis of these anomalies (as well as other structural characteristics) provides key insight into the types of malformations prevalent in the wild. Over the course of our development we exported results from our internal tools, and others, into a database from which we gained perspective on the malformations, correlated features, and synthesized conclusions.

2 Background

Let's begin by reviewing the basic abstractions involved in PE files. We'll delve more into the internal details of our parser later, but for now let's just review the basic PE-related concepts for reference purposes.

The PE file itself is, of course, the top level container for the entire PE representation. The PE file is comprised of a header, one or more sections (which we'll refer to as *PE sections* to disambiguate them from other section types, as the term *section* is heavily overloaded), and an optional *overlay* area.

2.1 Alignment and Mapping

The PE header specifies two alignment attributes which apply to the PE sections: *file alignment* is the alignment of the section's content on disk; *virtual alignment* is the alignment of that content in memory. Historically, file alignment has been smaller than virtual alignment, resulting in a more compact disk image.

As is probably obvious, PE images can exist in two different states: file mapped or memory mapped. We use the terms *file mapped* and *virtual mapped* to refer to these states, respectively. The source representation of virtually all PEs is file mapped, since this is the normative state of a PE when created by the linker. At some point, the Windows loader maps a file image into memory and creates the virtual mapping, which nominally pads out the sections to page boundaries (i.e., performs a virtual alignment), and discards any overlay. We'll use the generic term *image* to refer to a PE-formatted data stream, irrespective of a particular mapping.

2.2 Mapping Translation

The preceding discussion of image mapping leads to the realization that, since the PE image can have several representations, we might be able to devise a mechanism for transparently transforming between them.

More specifically, in a commercial application of this technology, we are dealing with not just file mapped (i.e., disk based) images, but virtual mapped images as well. For example, during a full system scan, a security product will iterate over the file system and process a number of file mapped images. It will also likely scan all running processes and examine the virtually mapped images as they exist in memory. These considerations make it desirable for our parser to include some means to accept either file- or virtual-mapped source representations. We've found it convenient to encapsulate the logic for dealing with all details of image mapping and translation into a separate component. This component provides general purpose functionality for accepting a source image in an arbitrary representation, translating it to an arbitrary representation for analysis purposes, and translating back (after remediation) if needed. We'll discuss implementation details later in the paper.

2.3 Size Matters

A basic and important abstraction to discuss is size, specifically in the context of PE sections and images, due to the varying ways in which these objects can be mapped. When discussing the "size" of these objects, we must be very careful to qualify whether we mean raw, file mapped, or virtual mapped size. *Raw size* refers to the nominal, unpadded size of the data; *file size* refers to the file-padded size, and *virtual size* refers to the virtually-padded size. For example, when searching a virtually mapped PE section, we may wish to search only the subset of the section corresponding to the file mapped part of the object. Although it sometimes seems

burdensome, it's important to be consistently precise when conveying size information.

It's both interesting and annoying to note that the PE format provides no direct, unequivocal specification of the "raw" size of the PE sections, although sometimes it can be inferred. In practice, the file aligned size is often treated as synonymous with the raw size of the image. In some cases, though, the distinction can be important, for example when dealing with malware that uses the slack space in the file alignment region to store nefarious information.

2.4 PE Header

The PE header is comprised of well-documented data structures that describe the organization of the PE image. The header structures are all generally located at the beginning of the image and provides all the descriptive information needed to parse the explicit structure[1] of the PE. Instead of reinventing the wheel, we'll simply refer to [1] and [4] for details for those who'd like to review them.

2.5 PE Section

The PE sections found in an image are described in the section table structure found in the image's header. The ordering of sections is usually the same in the file mapped image (a.k.a. the disk file, as given by the SectHdr.FileOfs field), the virtual mapped image (as given by the SectHdr.VirtAddr field) and the section table, although the PE specification does not require this. Although rare, we do see files whose section table entries are not ordered according to increasing file offset[2]. However, the section table entries always correspond to strictly increasing virtual address order [4].

2.6 Overlay

The overlay refers to any data appended beyond the nominal end of the PE image. This is data closely associated with the image, but generally not part of the content generated by the linker at file creation time. Overlays are usually created during post-processing by an external tool like a packer, or, in cases of malware, may be appended to the file (e.g., during infection). Common examples include certificates and debug information.

There is no requirement that overlays be homogeneous content; an overlay could be a composition of a certificate, debug information, and custom data appended by a packer, for example. We'll refer to homogenous blocks of data within the overlay as overlay *regions.*

Our parser provides functionality to isolate and enumerate the various overlay regions present in an overlay. Parsing this data precisely is something of a challenge because of the padding involved with the various regions.

2.7 Metasections

Abstracting the principal PE structural components described above leads to a significant simplification of the component model. We call the abstraction a m*etasection*, and in our implementation it serves as a base class for the PE header, PE section, and overlay. This model isn't perfect because the overlay does not follow the normal alignment rules that a PE header and section does, but on balance we've found the abstraction useful.

---

[1] That is, the portion of the PE that is "well documented". It's another problem entirely to parse code and data sections because (in the absence of symbols) these regions are opaque and lack any information as to component boundaries.

[2] Anomaly bits 10 and 11, respectively.

While an overlay is not loaded into memory by Windows at image load time, it may contain valuable information for purposes of analysis. Therefore, our parser treats the overlay like any other section and aligns and loads the overlay data in its own metasection[3].

Using metasections allows our parser to maintain a simple, ordered collection of non-overlapping instances corresponding to the entire source (input) image stream. Since the metasection abstraction aggregates a lot of common attribute information, the object model is significantly simplified.

2.8 Metadata

Our *metadata* abstraction refers to special, predefined data types that are enumerated in the Data Directory. Again, we found it convenient in the implementation of our parser to abstract this concept.

2.9 Caves

Suppose we have a virtually mapped PE image object. As we've defined it, the raw size of its code section is the nominal content present in the section; usually this corresponds to *unpadded* content written by the linker[4]. At file creation time, the linker rounds this raw size up to the file alignment of the section; we refer to the resultant padding region as a *file cave*. Assuming the virtual alignment value is larger than the file alignment value, when the section is virtually mapped, another alignment region may be created, which we refer to as a *virtual cave*. When dealing with caves, remember a key difference is that the file cave contents persist, because they exist in the file image, whereas virtual cave contents do not persist because they are created at image execution time and exist in memory only.

---

[3] This requires that certain fields in the data directory be updated to account for the padding injected into the target image.

[4] Determining raw metasection size requires heuristics, since this information is not directly available in the PE header. Raw size is useful in certain cases when performing in depth static analysis.

2.10 An Example

Let's put some of these concepts into use by looking at a real world example. The following information was generated from an analysis of a W32.GhostBot sample:

```
          +------ File Offs -------+ +---------- RVA ----------+ +---------- VA ----------+
 Section   Start     End      Size    Start     End      Size    Start     End      Size   Contents
 -------- -------- -------- -------- -------- -------- -------- -------- -------- -------- --------
 [header] 00000000 000003ff 00000400 00000000 00000fff 00001000 00400000 004003ff 00001000 PE hdr
     code 00000400 00000400 00000000 00001000 00011fff 00011000 00401000 00400fff 00011000 idata
     text 00000400 0000a7ff 0000a400 00012000 0001cfff 0000b000 00412000 0041c3ff 0000b000 idata, tls
    .rsrc 0000a800 0000b3ff 00000c00 0001d000 0001dfff 00001000 0041d000 0041dbff 00001000 idata, imports, resources
 [ovrlay] 0000b400 0000b4fe 000000ff 0001e000 0001e0fe 000000ff 0041e000 0041e0fe 000000ff Unknown
                           --------                  --------                  --------
Total Size                 0000b4ff                  0001e0ff                  0001e0ff
```

This sample contains 5 metasections: the header, 3 PE sections, and an overlay. The file alignment (not shown) is 0x400, and the virtual alignment is 0x1000. Note right away that section "code" has a file size of zero, which seems strange given that its characteristics indicate it contains initialized data. Figure 1 graphically depicts the organization of this image based on its structural information. In particular, note the file mapped representation contains a void corresponding to the virtual cave region in the virtually mapped image.

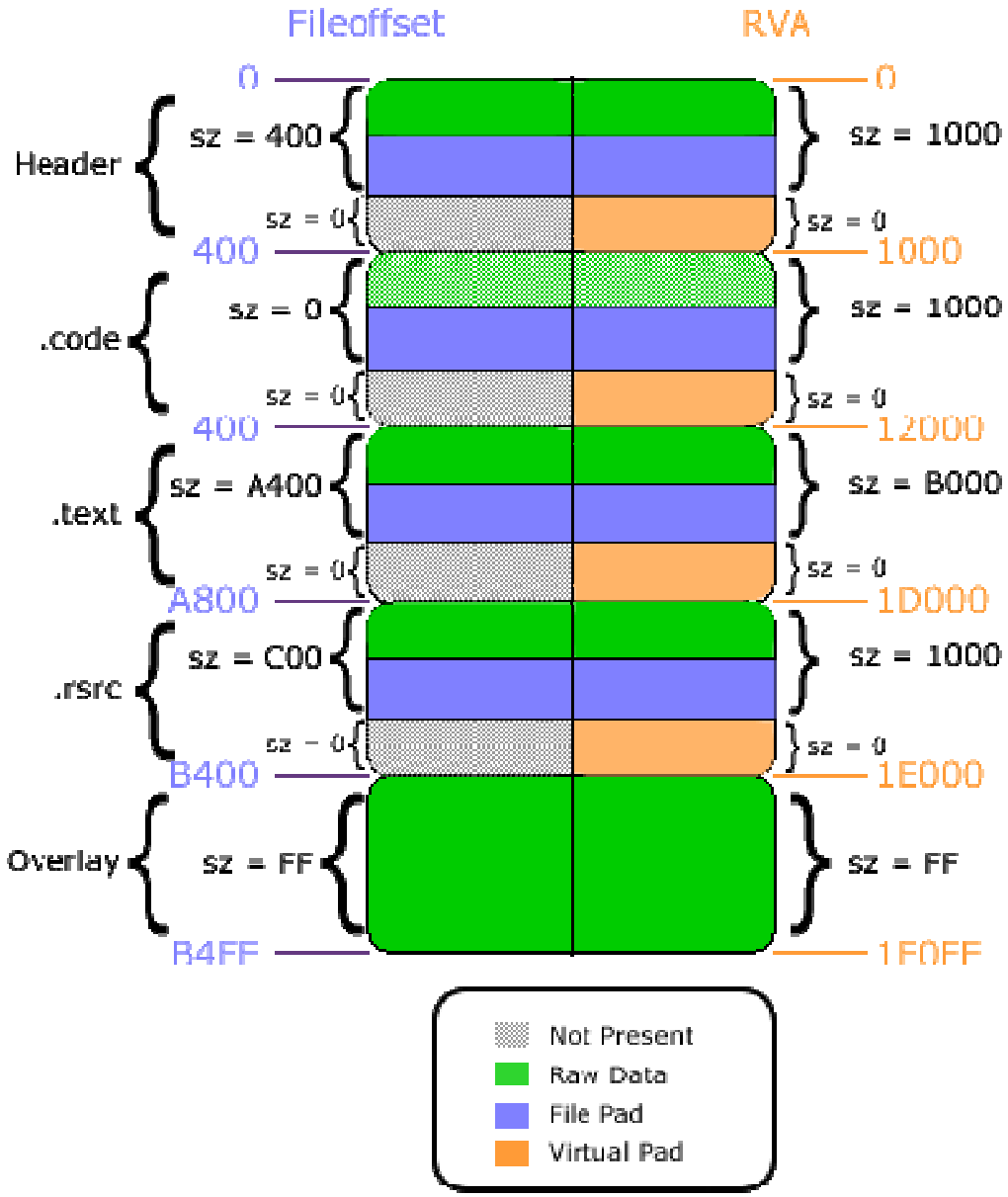


Figure 1: Sample image organization, file vs. virtual mapped layouts

3 PE Collection Analyses

Once we realized we would have to take an incremental approach to parsing our malware collection, the decision to organize the information in a database came naturally. In addition, we use a Design for Test development methodology, which demands regressions are built into the development process from the design phase onward, so we realized we would need to capture and store regression results. We built an extensive regression suite that verifies all aspects of our parser's behavior, and results are archived in the database. The database also supports research activities, including heuristic development, data mining, etc.

3.1 Image Analysis Process

Our process consisted of building several large datasets of source images; we then processed them through various tools, post-processed the resultant output, and imported it into a database. We then mined the database for interesting correlations, some of which were used in the development of our heuristics.

The datasets were comprised of a white list and a black list. The white list contained 8,959 known-good PE files; the blacklist contained 69,607 known-bad PE files. The toolset for this test was comprised of just two tools: PEiD [9], a publicly available packer identifier, and PeSweep, an internally developed tool and Swiss-army knife for testing our PE parser library.

PeSweep allows us to recurse over a directory tree and process all members through our parser, and perform other optional analysis steps as well. Depending on the configuration, a large amount of information can be generated for each executable, including:

- Whether Windows is able to load it[5];
- Details on how much of the structure the parser is able to parse;
- Entropy values on a sectional basis;
- Specific values from header structures, including the ability to normalize the header;
- The ability to create both file and virtually mapped target mappings of the image;
- Results of parsing metadata content, such as import, export, relocation, and resource values;
- Anomaly bits

The analysis phase of our work was time consuming, owing to its speculative and iterative nature. The results presented here are just the tip of the iceberg; we expect the quality of our results will improve over time. But clearly this is a rich repository of information, and a valuable resource to support ongoing R&D.

3.2 Analysis Results: Sampling Procedures

We have only scratched the surface of what is possible with our analysis activities so far, and the results seem quite promising. Let's take a tour of some of the more interesting data.

First we will provide some general results. In the subsequent discussion, the terms "white file" and "black file" refer to files from those respective collections.

Figure 2 shows how the number of PE sections varies through white and black files.

[5] More accurately, whether we *infer* Windows can load it based on results from other APIs

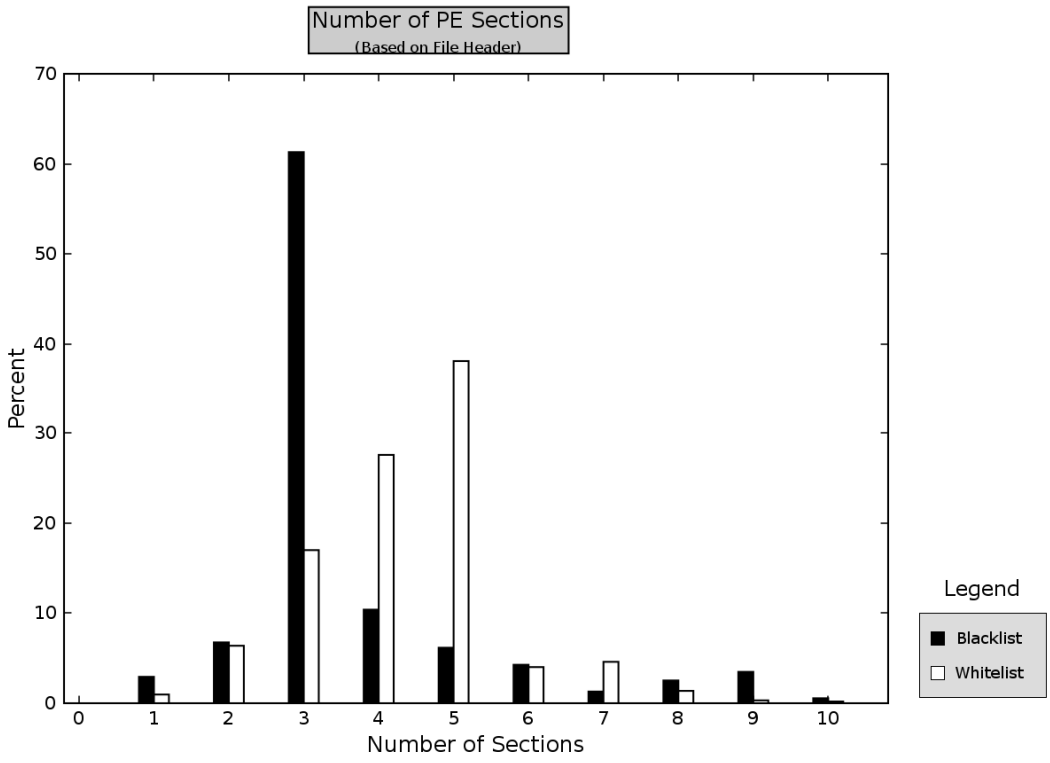


Figure 2: PE Section Counts by White and Black Files

This shows an interesting fact about black files in particular. They have a greater tendency to be packed, encrypted, or hand-crafted, which leads to a smaller number of sections in the image.

In order to fully understand this section count analysis, we've also included the maximum entropy values for each image, where entropy was calculated by finding the highest byte-entropy value of a PE section in each image. We discovered that malicious files generally have higher maximum entropy than their white-listed counterparts, as Figures 3 and 4 show.

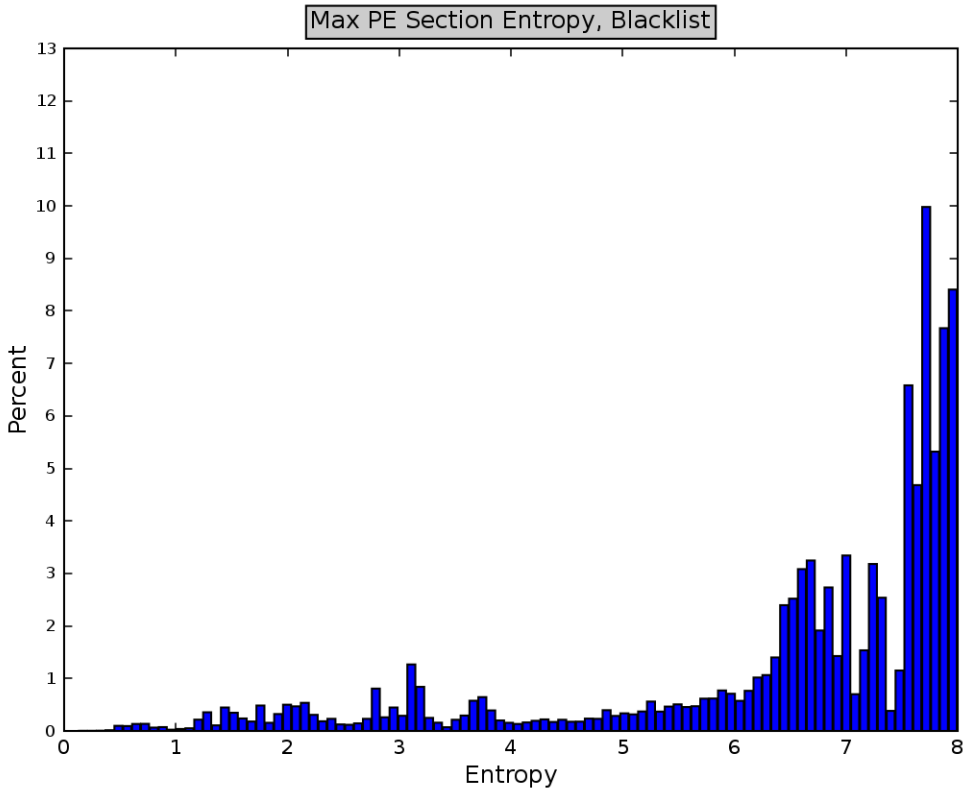


Figure 3: Maximum Blacklist Entropy

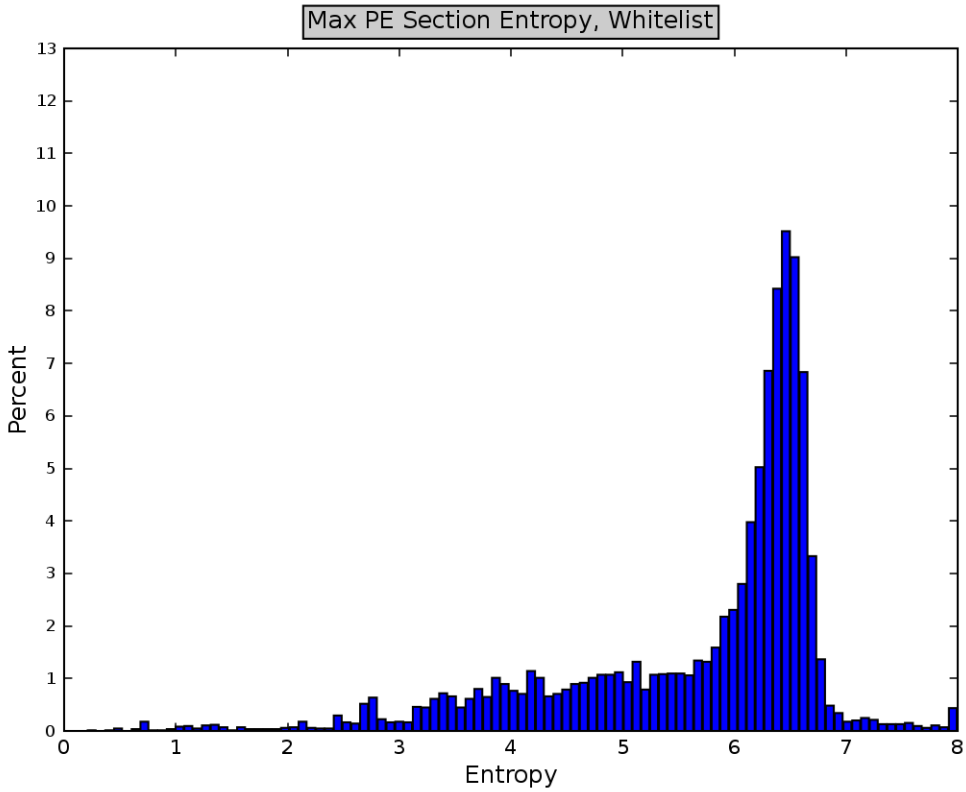


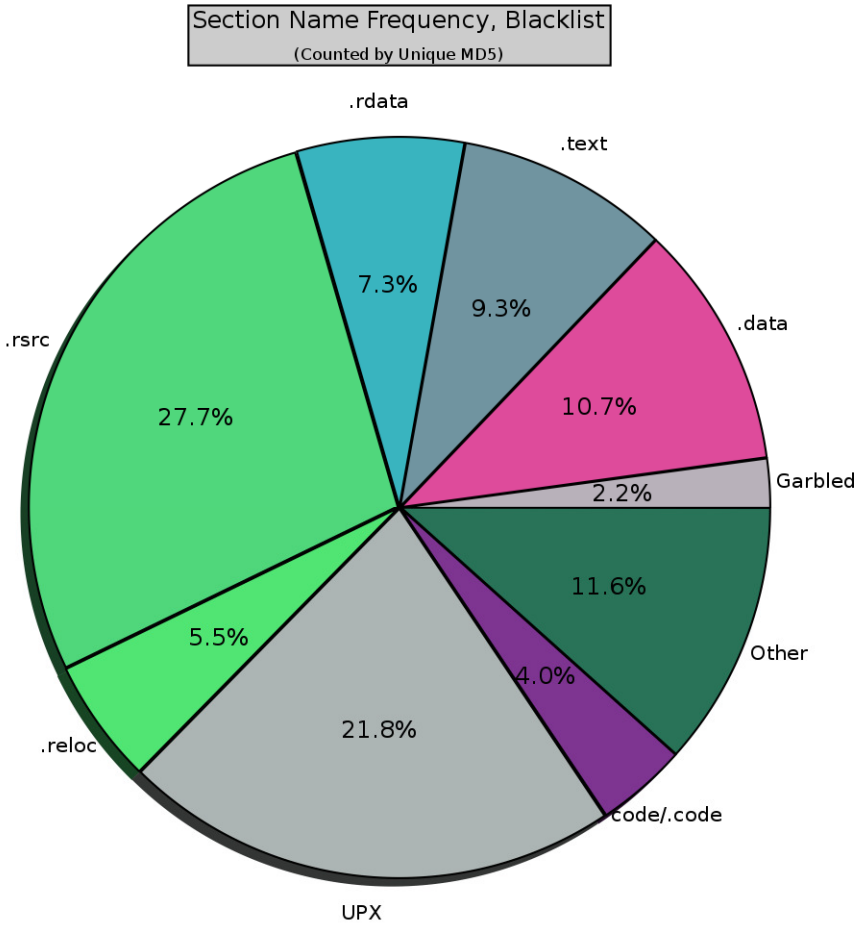
Figure 4: Maximum Whitelist Entropy

We also discovered a high correlation between black files with three sections and maximum entropy, which isn't surprising considering that packers like UPX arrange the output executable in three sections.

In addition to this data, we also tracked the section names for each PE. We found the following when counting section names per file: while benign files almost exclusively use default Microsoft Visual C++ values, and malware samples do not follow this convention. Indeed, many samples of malicious files are written with MSVC. However, they are typically then packed, encrypted, or otherwise post-processed.

The irony of obfuscating the internal logic is these characteristics make for very compelling malware heuristics, which anti-virus vendors can use to identify malware with a high degree of confidence. The best malware authors "hide the malware in plain sight," while the remainder choose the path of least resistance, packing or encrypting their file, thus signaling that they have something to hide.

Our analysis of sectional names covers two cases: the case where each section name is counted once per PE file, and the second case where each section is counted each time it appears. The differences are noticeable in blacklisted files, and signal the fact that malware authors, in addition to obfuscating various header characteristics, have a penchant for reusing a section name multiple times. In fact, some samples we've seen had the same section name listed ten times in a row, which is not only comical, but almost certainly malicious.


Section Name Frequency, Blacklist
(Counted by Unique MD5)
.rdata
.text
.data
Garbled
Other
code/.code
UPX
.reloc
.rsrc
7.3%
9.3%
10.7%
2.2%
11.6%
4.0%
21.8%
5.5%
27.7%

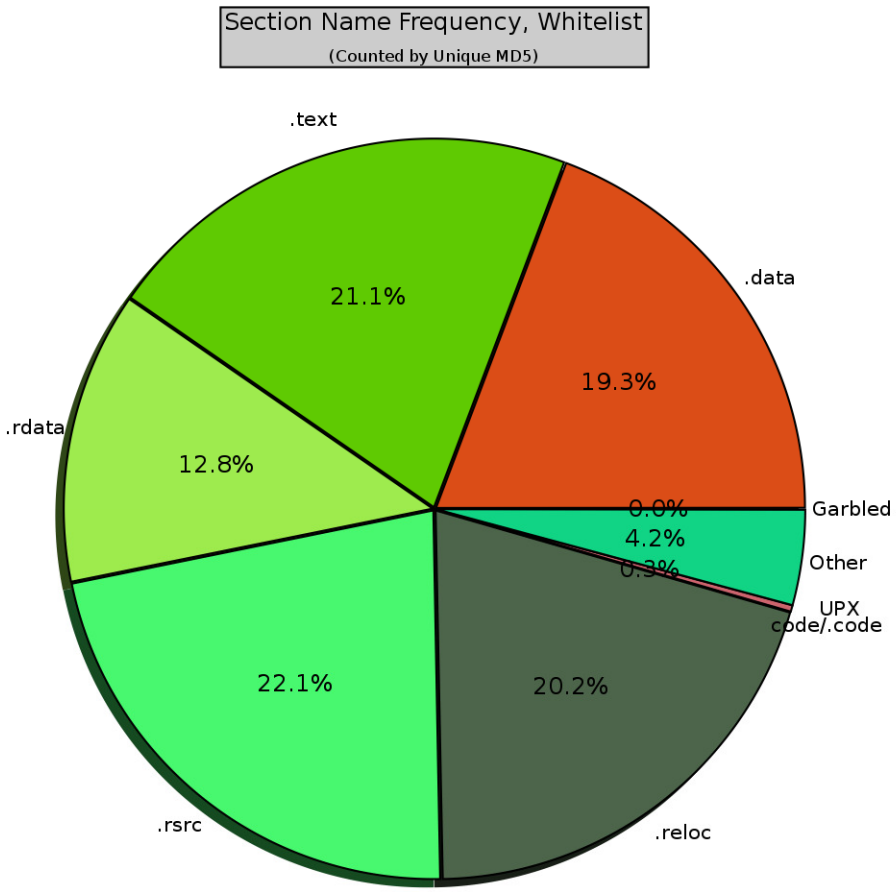


Figure 5: Section Naming

We also analyzed by sectional characteristics, which express the actual traits of the PE section. We came to some interesting, general conclusions regarding the normative structure of files. White files are typically organized with Read-Only sections following Read-Write sections, Data sections following Code sections, and Uninitialized Data following Initialized Data sections.

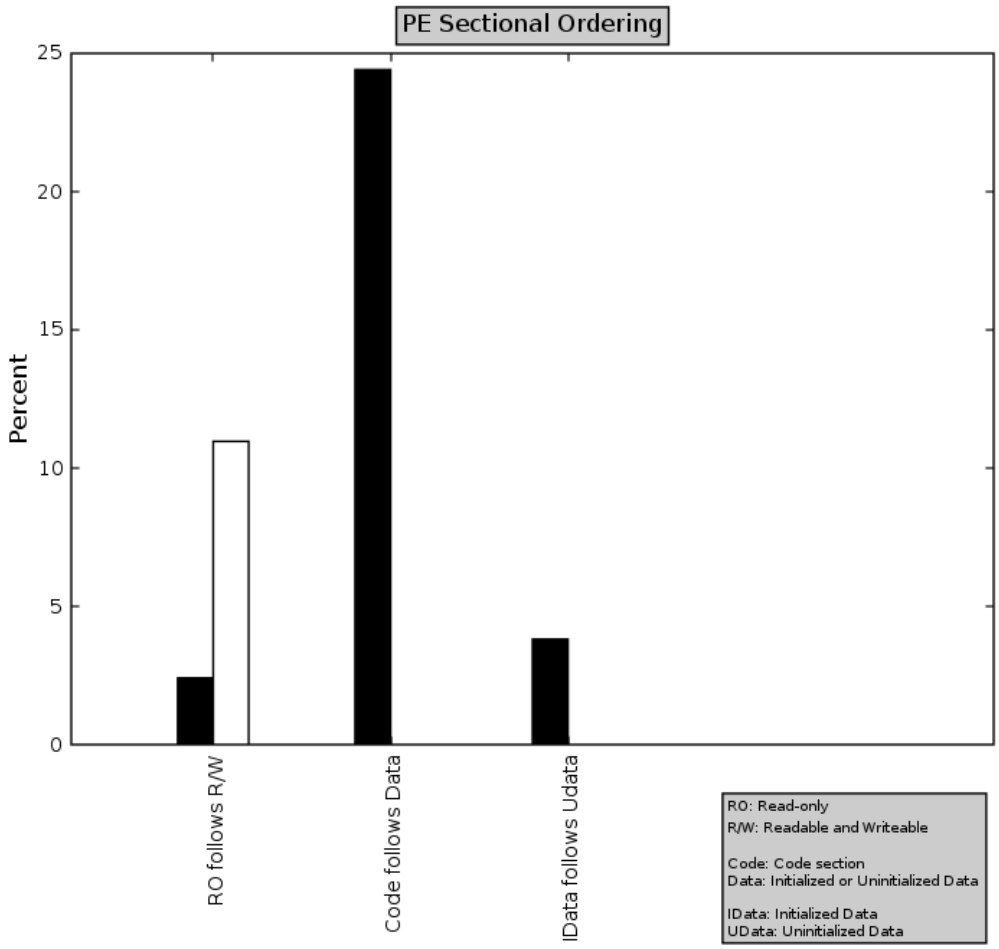


Figure 6: Sectional Ordering

Beyond sectional analysis, we also performed analysis of file metasection data, the most interesting of which is the overlay. Overlays expressed interesting characteristics across malware samples. In particular, a comparatively small number of benign PEs actually have an overlay section (approximately 30%), whereas twice as many blacklisted files had at least one overlay subsection.

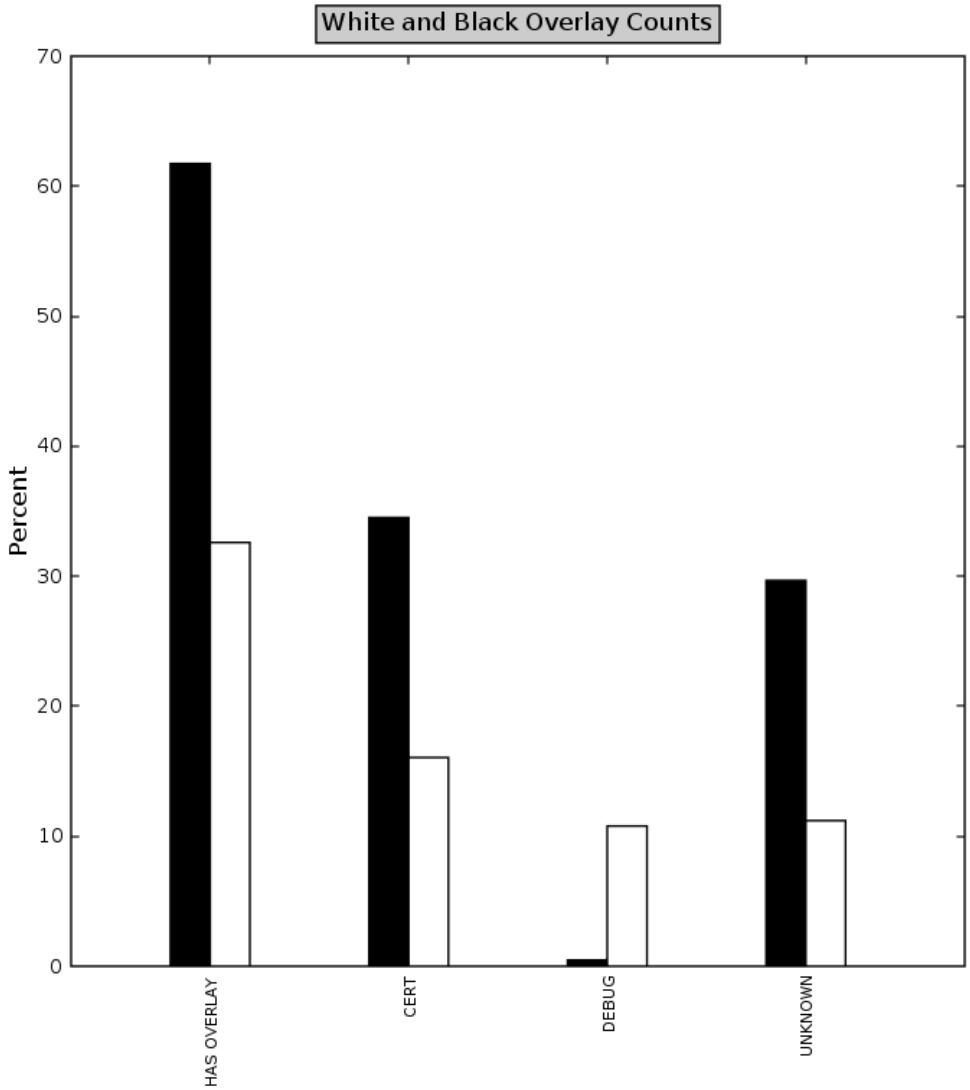


Figure 7: Overlay Counts

Interestingly, while certificates are very commonly used, debug sections were not. Also of note are our "unknown" overlay types which indicated content we cannot classify.

Finally, we analyzed and graphed the various anomaly bits defined by our parser. Figure 8 shows the prevalence of each anomaly bit, normalized by relative frequency.

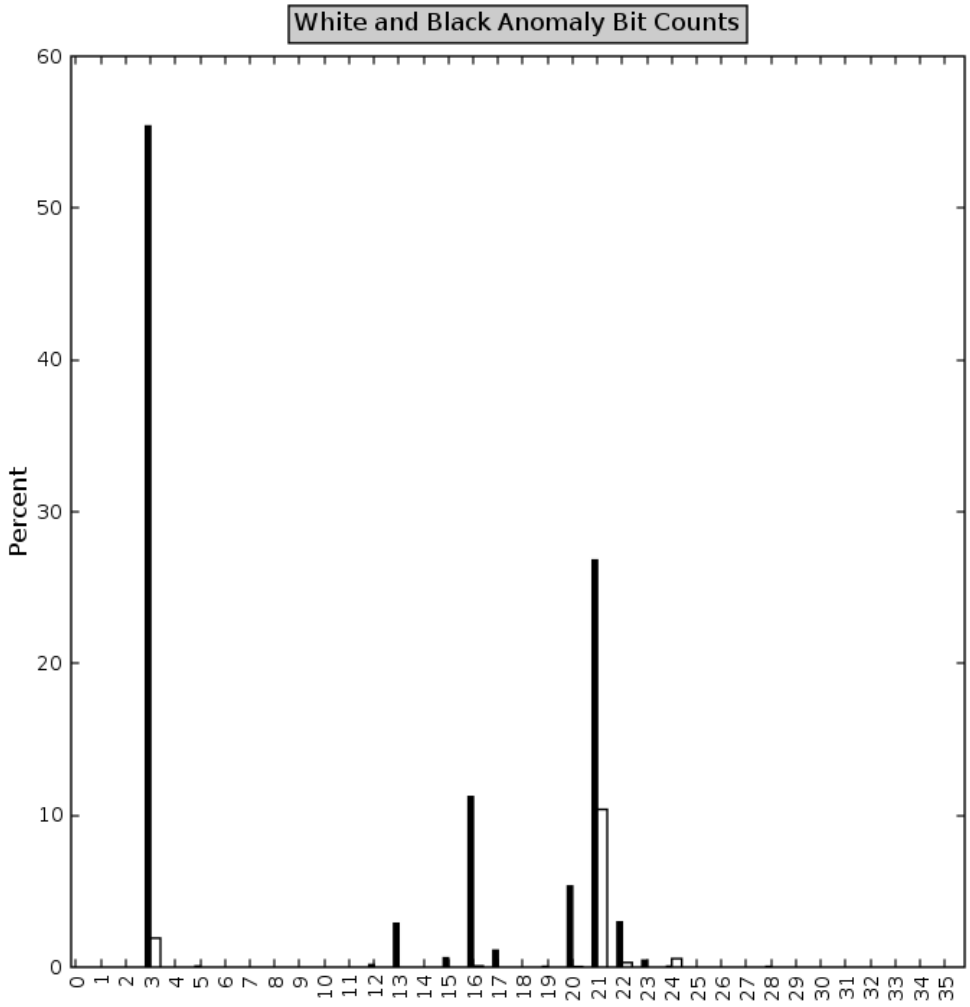


Figure 8: Anomaly Frequency

Each anomaly bit (x axis) is associated with a specific malformation. When the parser parses through a PE file and finds a malformation it flags the corresponding anomaly bit. These results show some very high anomaly counts across our entire sample set between white and black files. Because of this, many of these anomalies would make very good heuristics. Descriptions of the notable bits are listed below:

Bit 3 – The SizeOfHeader field is unaligned
Bit 13 – DOS header and optional header are overlapping
Bit 16 – The last section header's size is not aligned
Bit 17 – A size in the section header is too large
Bit 20 – Section name garbled (non-printable characters)
Bit 21 - There is an unknown overlay region
Bit 22 – Zero/nonzero pair in DD entry
Bit 23 – Pointer too large in DD entry

Bit 24 – Size field in section header not correctly aligned

3.3 Utilizing the Data

There are a variety of interesting applications for this analysis data, and we want to discuss two of them in particular. First, we're interested in determining at parse time if a target executable is a valid PE (in terms of whether Windows will load and run it). This determination allows us to filter executables that might be mangled so badly that they are not runnable[6]. The problem reduces to creating heuristics, based on anomalies, which will identify malformed images with a high degree of confidence. These malformations are such that the image is not loadable and/or runnable by Windows.

To solve this problem, we queried our database for PE files (by the loosest measure: valid DOS and NT signatures). We then tracked, on four different versions of Windows (NT, 2000, XP, Vista), which files our parser could successfully parse, and compared those to results from certain Win32 API calls from which we can infer loader behavior: CreateProcess and LoadLibraryEx.

Of these two APIs, LoadLibraryEx is the most flexible, as CreateProcess will refuse to load an otherwise valid PE with the `IMAGE_FILE_DLL` flag set (i.e., a DLL), while LoadLibraryEx will attempt to load stand-alone executables, DLLs, and drivers. LoadLibraryEx has its own problems however; for example if relocation fails, LoadLibraryEx will also fail.

Due to these considerations we synthesized a "loader test" using both APIs; if either CreateProcess or LoadLibraryEx is successful, we consider the image loadable. Figure 9 depicts the results of our testing, and shows a high correlation between the operation of the Windows loader, and our own parser. This is reassuring as it indicates a high degree of agreement between the Windows loader and our parser.

The second potential application of analysis data is to determine if a target file is malicious. Clearly a heuristic generating a highly reliable "Is Suspicious" flag is very valuable during image parsing. With a single, simple query of our database, we are able to identify 67% of the black-list with a false positive rate of just 1.4%. The query checks anomaly bits only, and could be improved significantly by incorporating additional heuristic data.

[6] We're uncertain as to how useful this will be in practice; it may be more valuable as an aid in classifying images in our internal blacklist repository, many of which appear to be badly mangled.

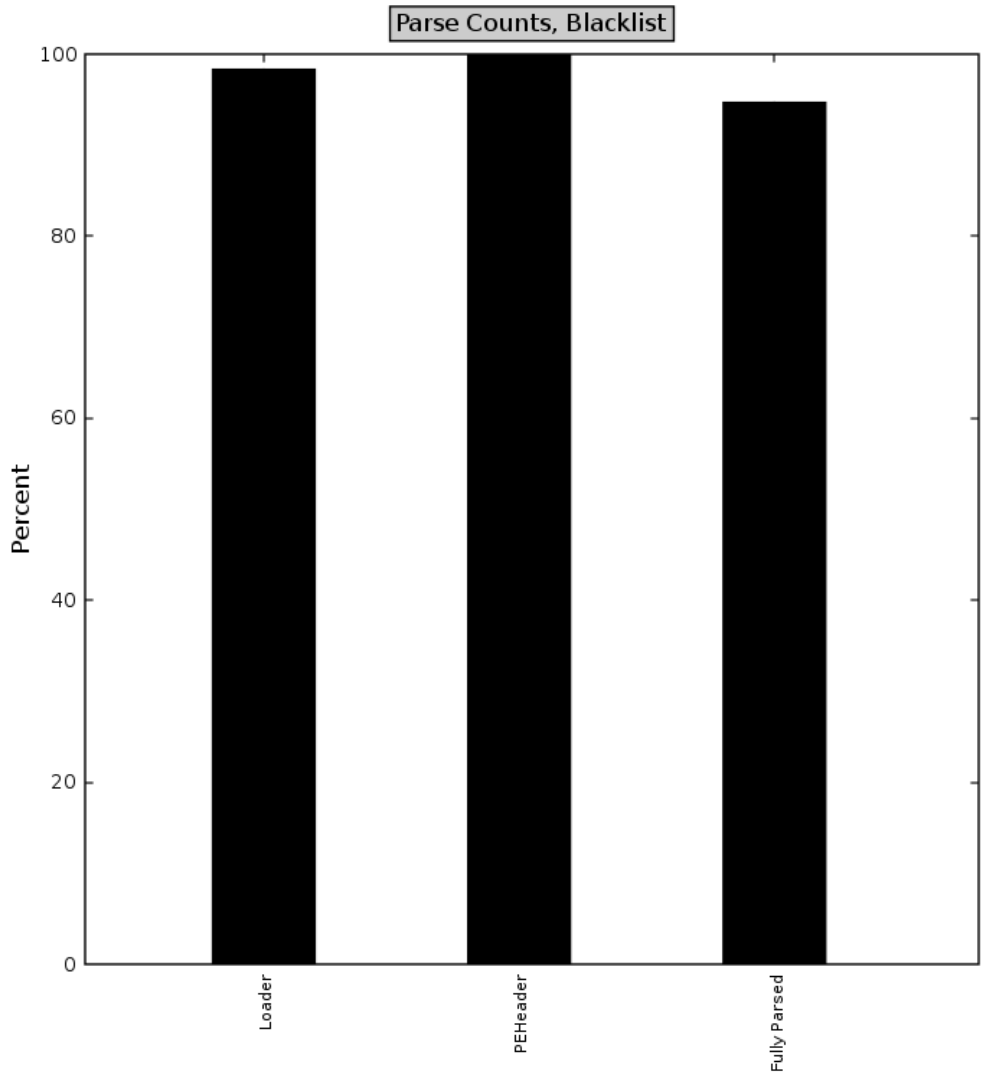


Figure 9: Parse counts versus the loader, blacklist files

3.3 Summary

We've developed a general approach to managing our data that has proven helpful in several key ways:

- Ongoing refinement of our parser performance and quality,
- Management of our regression results, and
- Support for ongoing R&D activities

Key aspects of our analysis approach include:

1. Build a large set of representative white and black PEs.
2. Run a tool set (including the parser) against the data set to
    a. Establish baseline parser performance metrics
    b. Collect detailed static analysis information
3. Import results into database.
4. Verify parser performance has not regressed since last release.

5. Iteratively improve parser performance

4 Image Parsing In Depth

With all the background out of the way, we're finally ready to jump into the real meat and potatoes of PE parsing. We'll provide an overview, and discuss some of the design tradeoffs of our parser, then look at the parsing process in more detail.

4.1 Implementation Overview

Figure 10 shows the class diagram representing the major components in our parser library. In our implementation, cPeHeader, cPeSection and cPeOverlay are all subclasses of the abstract class cMetaSection; this arrangement simplifies certain component interfaces significantly. cMetaSection represents any aligned, contiguous region of data (or code) within the image.

cPeImageStream encapsulates all the logic required to size and map a PE image. It is capable of accepting a PE in any source representation (file- or virtual-mapped) and translating to any other representation. Since VIPRE dispositioning logic includes a generic unpacking facility, source images may be emulated, so we chose virtual mapping as our canonical representation.

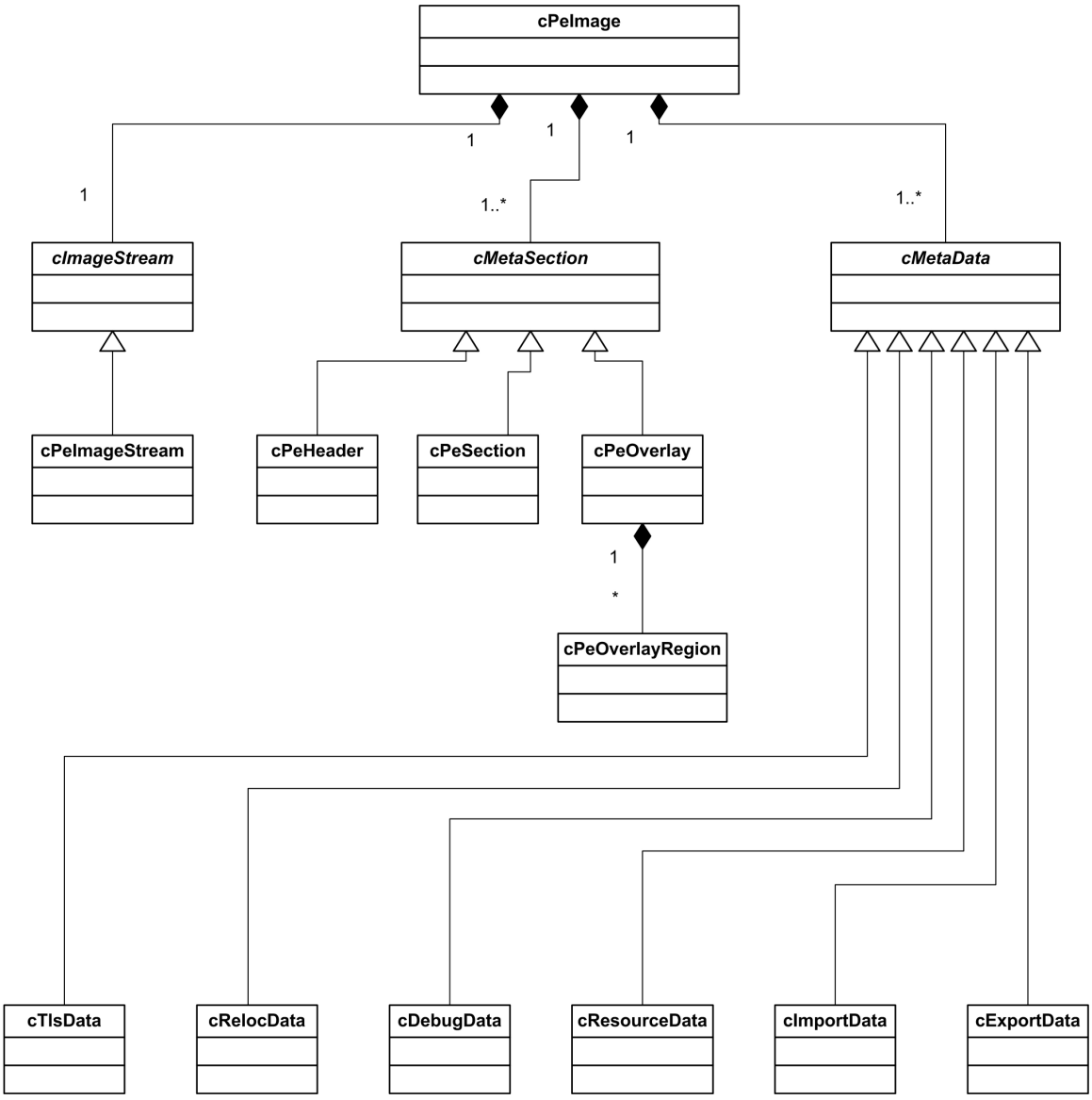


Figure 10: PE library class organization

4.2 Parsing An Image

The flowchart in Figure 11 provides a high level overview of the entire image parsing process. The input for this process assumes we have been given a stream-based representation of a PE image, and told how it is mapped. The basic logic involves creating a cPeImageStream instance, and then initializing it with the source stream. Assuming initialization completes successfully, the ImageStream contains a target stream, which is a properly mapped and loaded representation of the source stream. Note that a malformed PE, if present, is identified at this stage.

Next, a cPeImage instance is allocated, and then initialized with the cPeImageStream just constructed. Finally, all metadata present in the source image can be fully parsed by cPeImage, resulting in a completely initialized IR.

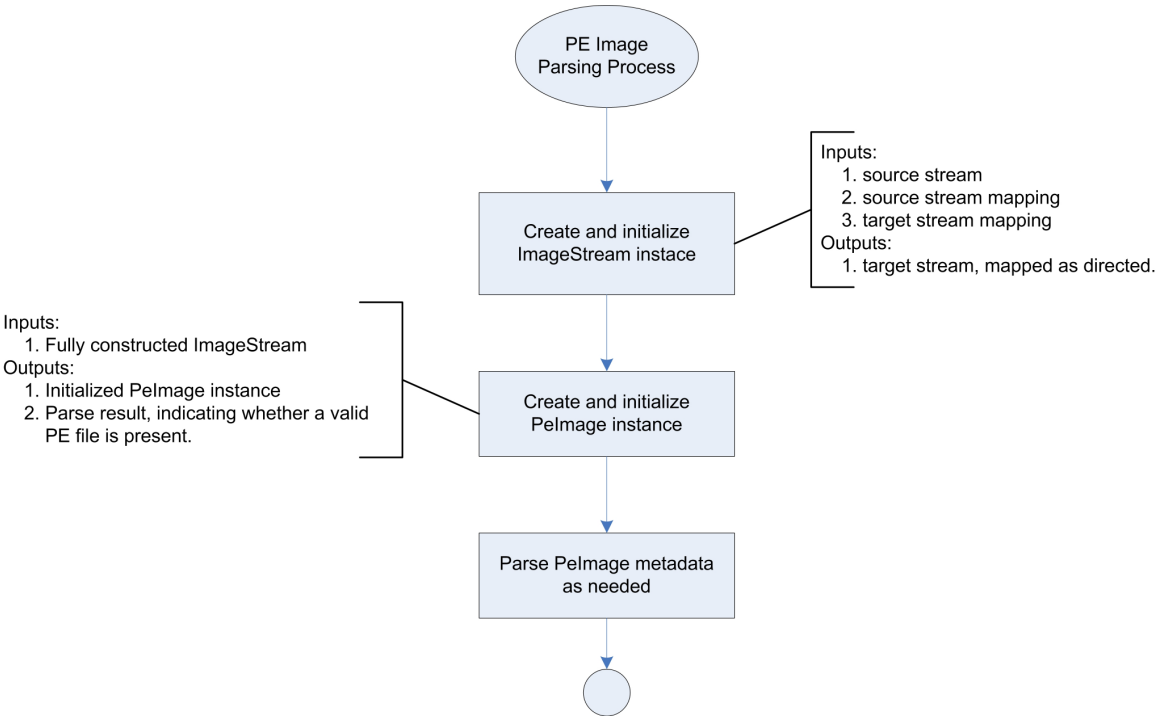


Figure 11: Top Level Image Parsing Algorithm

While some of the information contained in the header is required to be correct, the Windows loader appears to be very accommodating with respect to poorly formed structures. Figure 12 below is a list of key fields in the header, along with comments as to how they are treated by Windows.

Field Name	Comment
DOS Header	
Magic	Immutable
e_lfanew	Immutable, although the NT Header can be placed anywhere in the stream as long as this field has the correct file offset.
File Header	
Machine	Immutable
Number of sections	Immutable, but sections can be hidden in the header
Size of opt header	This is used in a calculation to determine where the section table starts, it does not have to reflect the actual size of the optional header
Optional Header	
Magic number	Immutable
Address of entry point	Immutable
Image base	This is used by loader but can be any value
Sect/File alignment	The value must be a power of 2, and as long the Section alignment is greater than the file alignment any value will work
Major subsystem version	Immutable
Size hdrs	Part of the calculation for the size of image
Size image	This field uses the size of hdrs field and the size of code

	fields to calculate the size of the image
Subsystem	Immutable, This is the last field in the header that is required
Section Table	
Name	Must be eight bytes in length, any value
Virt/file offset	These values must properly point to the start of a section

Figure 12: Interesting PE Header Characteristics

4.2.1 Creating an cImageStream

The cImageStream class incorporates the logic for creating the target stream, and is a convenient wrapper for all the information associated with the source and target streams. This makes life easy for the cPeImage class, because everything it needs to know about the source and target streams is wrapped up in a single class.

Figure 13 is a detailed flowchart for the ImageStream construction process.

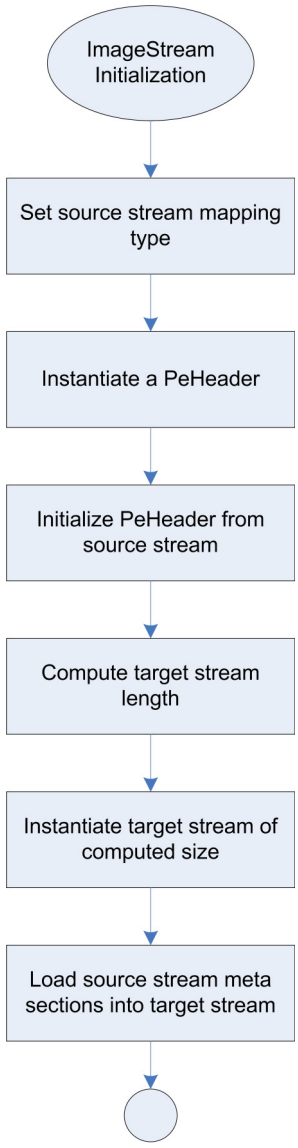


Figure 13: ImageStream Initialization Process

Parsing the Header

The parser first reads the source stream from disk or memory. It instantiates a cPeImageStream class and passes the source stream to it for initial parsing, performed by a cPeHeader instance. Throughout the entire parsing process, the parsing logic constantly checks for, and tracks any anomalous conditions that might be discovered in the structure of the source stream. Next, the cPeHeader class calculates the size required for the target stream by summing the sizes of the source stream's metasections, according to the target stream mapping.

After the proper size is calculated, the parser can begin loading each source metasection to the target stream. One after another, each metasection is placed into the target stream at its correct offset. This is a rather delicate process; many anomalous characteristics can create inconsistencies in the target stream. One possible anomaly occurs if a section is unaligned. When building the executable, by default the linker will align each section according to the file alignment field in the Optional Header.

If no errors occur, the parser will have created a fully mapped and loaded cPeImageStream instance comprising the source stream correctly mapped per the client's specification. The header is parsed and validated. In addition, the full range of anomaly information compiled during the process is made available.

Input Stream Normalization: It Seemed Like a Good Idea at the Time…

When dealing with malware, or for that matter any PE image, the parser can make no assumptions about the validity or consistency of any particular fields in the header. Images can be hacked or patched in myriad different ways. Any abnormal values increase the chances of erratic behavior (i.e., crashing) during parsing. In order to prevent this, we implemented a normalization technique that, initially at least, seemed to be quite reasonable.

The basic idea is to make an analysis pass through the header and correct any values that are obviously illegal or improper, in order to guarantee that the subsequent parse pass succeeds. The parser would analyze the header, and apply fixes for any abnormal values it encountered. Initial results from this technique looked very promising, and the number of blacklist files we were able to parse successfully jumped dramatically.

In hindsight, we've learned we need to be extremely liberal insofar as the application of boundary and format checks applied to the image, because Windows itself allows all kinds of malformed images to be loaded and run. A great example of this is TinyPE [7]. This particular sample doesn't do much, but it sure is a good example of the types of malformations we run into in the real world when processing malware, and also the problem with normalization in general. More precisely, normalization enforces a certain structural rigidity that is often more restrictive than the Windows loader itself.

TinyPE uses some interesting techniques for shrinking the file size, but much of what it is doing is simply overlapping unused areas in header structures. For example, the file ends before the nominal end of the Optional Header is encountered. TinyPE relies on the loader's behavior to cover its tracks, and the missing fields in the Optional Header are zeroed by the underlying memory page supplied by the loader. The section table overlaps the Optional Header after its first field, so these structures are effectively compressed. Also, the image's executable code is tucked into the header, and the NT Header overlaps unused fields in the DOS Header.

In summary, our normalization approach initially was overly restrictive in some cases, but by studying cases like TinyPE we have been able to relax its behavior.

Using what we have learned from our experiences with normalization, we've been able to fine tune our parser logic so that only minimal normalization is now required.

Mapping the Target Stream

The map and load process is conceptually simple. The desired mapping of the target stream must be specified: file mapped or virtual mapped. The mappings are similar, as described in section 2, and in some cases identical. If the mapping alignments are the same there will be no difference between the size of a virtual mapped and file mapped image.

Let's define terms here precisely. *Mapping* is the process of sizing and allocating a target stream; *loading* is the process of translating a source stream into the mapped region. The inputs to the mapping process include the source stream itself, the source mapping, and the desired target mapping. The outputs include a target stream, mapped according to the caller's specification.

Sizing the target stream is the first step necessary to properly initialize the cPeImageStream, because the target stream must be sized and allocated prior to construction of the mapped image. This relatively complex calculation is based on various sizes and offsets in the header, added together. Since the parser accounts for the possibility of an overlay, the overlay size is added into the calculation as well. Once this size is calculated, the parser has the information that is required to create an ImageStream.

Loading the Target Stream

Loading is presumed to follow mapping. Inputs to the loading process include the source stream and target stream. During the loading process, the source stream contents are copied into the target stream, according to the specified target mapping.

4.2.2 Creating a cPeImage

The cPeImage class is a convenient wrapper for the IR of the source image contents. The IR refers to detailed, highly organized collection of data generated by the parsing process. This includes content garnered from the header, PE sections, overlay, and all data directory items. The cPeImage class provides a rich API to access this data efficiently.

Figure 14 is a detailed flowchart for the cPeImage construction process.

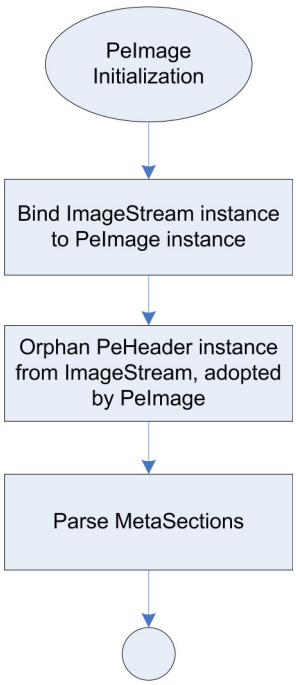


Figure 14: PeImage Initialization Process

Binding cPeImageStream

After the cPeImageStream is fully initialized, we create a cPeImage instance and bind the cPeImageStream to it. Since the cPeImageStream has already parsed the header and flagged any structural anomalies, cPeImage does not need to perform this operation again. When the cPeImage object gets allocated, the header is orphaned to it. cPeImage then uses this object to build a list of metasections, including the header, PE sections, and an overlay if present.

Parsing MetaSections

cPeImage provides a rich interface to access the header, which is now fully initialized. The next step is for cPeImage to walk the section table and instantiate instances of cPeSection, and a cPeOverlay (if an overlay is present). At the completion of this process, all metasections are represented in the cPeImage IR.

Parsing MetaData

The core parsing process is now complete, and we have all the basic information needed to describe the PE. However the process is not totally complete, because the metadata information has not yet been processed. The metadata structures provided by the PE include the import table, export table, etc (see Figure 10). This information is enumerated in the data directory. In our implementation, we have a large number of cMetaData subclasses which each parse a specific class of data. Not all of this information is always required, so it may be possible to skip certain of these steps, depending on the client's needs.

More information on metadata parsing is available in [1], including complete source code for Pietrek's PEDUMP utility.

Conclusions

This paper reviewed our strategy for developing robust and comprehensive PE image analysis capabilities, including facilities for regression testing and ongoing

R&D, which have provided our development team with many valuable advantages during the development of our security solutions.

We also presented insights into the design and implementation of a robust PE parser library, which we hope will help increase the understanding of this topic.

A key part of our effort has been to identify and track common image malformations, using the anomaly mechanism. The analysis of these anomalies (as well as other structural characteristics) provides insight into the types of malformations prevalent in the wild. In the course of our research, we categorized the results into a database, which helps us improve our tools, gain perspective on malformations, and correlate features.

References

1. Matt Pietrek, Under The Hood, An In-Depth Look into the Win32 Portable Executable File Format, MSDN Magazine, April 2002, http://msdn.microsoft.com/msdnmag/issues/02/02/PE/
2. Russ Osterlund, What Goes On Inside Windows 2000: Solving the Mysteries of the Loader, http://msdn.microsoft.com/msdnmag/issues/02/03/Loader/ MSDN Magazine, March 2002,
3. Matt Pietrek, Under The Hood, http://www.microsoft.com/msj/0999/hood/hood0999.aspx, Microsoft Systems Journal, September 1999
4. Microsoft Portable Executable and Common Object File Format Specification, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
5. http://upx.sourceforge.net/
6. Peter Szor, The Art of Computer Virus Research and Defense, Symantec Press, 2005
7. http://nepenthes.mwcollect.org/
8. Solar Eclipse, Tiny PE: Creating the Smallest Possible PE Executable, http://www.phreedom.org/solar/code/tinype/
9. PEiD: http://www.peid.tk/
10. Mark Russinovich, David Solomon, Microsoft Windows Internals, Fourth Edition, Microsoft Press, 2005