The Forgotten Stub: Exploring Malicious Use of the PE DOS Header

Sayonnha Mandal[1], Kshitiz Aryal[2] and William Mahoney[2]
[1]Metro State University, St. Paul, Minnesota, USA
[2]School of Interdisciplinary Informatics, University of Nebraska at Omaha, USA

sayonnha.mandal@metrostate.edu
karyal@nebraska.edu
wmahoney@unomaha.edu

Abstract: Malicious actors increasingly employ sophisticated concealment techniques within Windows Portable Executable (PE) files to evade static and dynamic detection, complicating incident response and digital forensics. Detecting malware in PE files has become a central challenge in modern security research, leading to a mix of complementary analysis methods. Current approaches range from traditional signature-based scanning, which identifies known byte patterns, to heuristic systems that flag unusual structural traits such as abnormal section sizes, entropy spikes, and inconsistent header values. Machine learning models now play a role, using features like opcode sequences, imported API functions, and metadata patterns to classify files at scale. Deep learning models, including convolutional and recurrent networks, learn higher-level representations directly from raw binaries or extracted features. However, many or most systems designed to detect malicious software in PE files deal with the portion of the file structure specific to Windows. A section of the file called the "DOS Header" is generally ignored by malware analysis. This paper describes a method whereby hand-crafted malware can be hidden in the DOS Header of the PE file, thus evading detection by many analysis methods.

Keywords: Reverse engineering, Malware, Windows, Software exploits

1. Introduction

Malware analysis is the practice of studying software to understand what the software does, how the software does it, and if the software is malicious, how it operates and how one might defend against it. Two methods for analysis are static inspection, where code and metadata are examined without running the file, or dynamic inspection, executing the sample in a controlled environment to watch its actions. These methods reveal the strategies malware authors use to hide, spread, and cause harm. By uncovering these details, malware analysis supports everything from incident response to the design of stronger detection tools, forming a key line of defense in modern cybersecurity. However, in the world of Microsoft Windows, many, if not most, software analysis tools focus on the portion of a Windows executable program that contains the Windows-relevant software. Because of the evolution of operating systems from Microsoft, executable program files on Windows contain a short "stub" program; this was designed during the transition from old MS-DOS to new Windows operating systems, and is actually a DOS program "stuck on the front" of the Windows instructions.

Can a malware author craft exploit code and hide it in the "stub" program, so that most modern vulnerability analysis tools will miss it? Yes.

In the following section, the authors describe the structure of a Windows "Portable Executable" (PE) file and detail the organization of the sections and headers in the file format. Section 3 provides a brief description of the tools commonly used in the reverse engineering process, where malware analysts inspect a PE file to determine how it operates. Section 4, "Current Trends in Malware Detection", covers exactly that – how Artificial Intelligence and Machine Learning, among other methods, are changing the landscape. Section 5 describes the author's methodology for hiding malware in a way that evades many of the tools that focus on Windows. Conclusions and final thoughts are in the last section, Section 6.

2. The PE File Format

The Windows Portable Executable (PE) file format is the standard file structure used for object code, executables, and dynamic link libraries (DLLs) in modern Windows operating systems (Microsoft 2025). A large part of the PE file format is taken from an older format called the Common Object File Format (COFF). In fact, the majority of the "interesting parts" in a Windows PE file are in the COFF portion of the file. A brief overview of the format of Windows PE files is in order.

A PE file consists of several parts, all of which are described by "headers". A header is simply a structure, or record, which describes a portion of the remainder of the file. Several of the headers are used to describe the entire contents, while some headers contain a set of records, and the collection of these records makes up the header. Figure 1 below gives the overall idea, and following the figure are additional details.

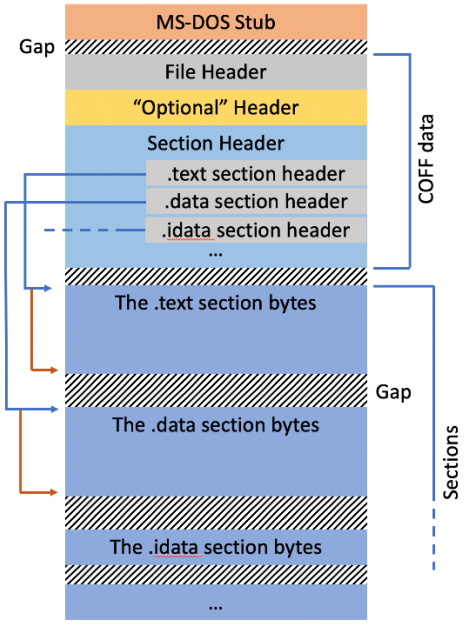

Figure 1: Typical Windows PE Executable File

Starting at the beginning (top) of the figure, and for backward compatibility, the beginning of the file is a program that can execute on a MS-DOS operating system. This DOS header is a minimal program that, when run on MS-DOS, will simply print a message and exit. At the time Windows was introduced, an MS-DOS program had a name ending in ".EXE" and a very simple format called the "MZ" header dictated what MS-DOS was to do with the remainder of the file. With the introduction of Windows, it was possible for a person to take a Windows executable file, also ending in ".EXE", and copy it from a Windows computer to a DOS computer and then attempt to run the binary. The "stub" on the front is present for this purpose and is a valid DOS program "stuck on the front of" the Windows binary.

Within the "MZ" header is a four-byte value that indicates the beginning byte of the COFF portion of the file. While the DOS portion starts with "MZ", the COFF portion starts with "PE", the first two bytes of the "file" header. The purpose of this header is to indicate what type of machine will run this code, how many sections (below) are in the file, and the size in bytes of the following "optional" header. Additionally, this area contains flags to indicate whether there is debug information, the byte ordering of the data in the file, and other characteristics. In a nutshell, this part of the file contains an overall view of what is coming up in the remainder.

Next in the file is the "optional" header. The quotation marks around the term "optional" are intentional, as the header is, in fact, required for executable programs. Despite its name, the Optional Header must be present in executable files; this naming convention originated from Microsoft's design of the format. Included are settings to indicate whether the program is a regular image or something else. A value in this area also indicates the number of bytes of program instructions, the number of bytes of initialized and uninitialized data, and so on. The address in memory for the first instruction in the program is also here, as is the base address where the program needs to be placed in memory.

It is important to understand a few additional values that are in the "optional" header. The "section alignment" is a number, a power of two. When a section (below) is loaded into memory, the address where it is loaded will be evenly divisible by this quantity. Similarly, the "file alignment" says that each section in the file will be at a byte position divisible by this alignment. Typical values might be 512, 1,024, 4,096, and so on. Note that since the following sections of the file must be located only at certain points in the executable, there are gaps between the sections, as shown in Figure 1 above.

The file is then considered as multiple sections, generally including the ".text" section for program instructions, ".data" for initialized data, ".rdata" for read-only data, possibly a ".bss" section, and others. Each of the section headers describes where in the file the section data starts, and the length – how many bytes. Additionally (not shown) are flags that indicate what Windows should do with the section; is it loaded and executable, for example, and where the section needs to reside in memory when it is loaded. Each section starts in the file at a location dictated by the file alignment described previously. For sections that are to be loaded in

memory, then, Windows will read the data from the file starting at the first byte of the section and through the length of the section, rounded up to the file alignment. A ramification of this is that the bytes in the file gaps are loaded into memory as well. These gaps do not contain anything of interest.

Unless they contain malware…

Overall, the PE format acts as a bridge between compiled code and the Windows loader, informing the loader which portions from the file go into which areas of the memory.

If the Windows program contains malware, various tools exist to assist in the analysis of that malware; some of these tools are described next.

3. Background on Reverse Engineering of Binary Executables

Reverse engineering binary code largely centers on disassembling and reconstructing machine instructions into a form that humans can understand. The difficulty of this process depends heavily on the system's architecture. For example, in architectures like ARM and other Reduced Instruction Set Computers (RISC), the consistent instruction length and predictable memory alignment make disassembly more straightforward. In contrast, Complex Instruction Set Computing (CISC) architectures such as the x86 family pose greater challenges due to their variable-length instructions and irregular encoding. Additionally, the vast number of instructions—some undocumented or dependent on specific contexts—further increases the complexity of reverse engineering (Mahoney, 2021).

Reverse engineering typically combines both static and dynamic analysis techniques. Static analysis focuses on examining program instructions without running the binary, with the goal of identifying properties that remain consistent across all possible execution paths. Common methods include disassembly, decompilation, control flow graph (CFG) reconstruction, and data flow analysis—all used to extract structural and behavioral information from the code. Well-known static analysis tools include Ghidra (2025), objdump (LinuxDevCenter, 2025), and udcli from the Udis86 project (2025).

In contrast, dynamic analysis involves executing the binary to monitor its runtime behavior and gather execution data. This approach is inherently limited to the code paths that are actually executed, which means it may overlook conditional branches or untriggered logic. Popular dynamic analysis tools include Binary Ninja (2025), Hopper (2025), OllyDbg (2025), Relyze (2025), and x64dbg (2025).

IdaPro (Hex-Rays 2025; Pearce, 2008) serves as a hybrid tool, combining static disassembly and analysis with integrated debugging capabilities to support partial dynamic analysis. Additionally, online services such as Defuse (2025) provide lightweight, web-based disassembly interfaces. Collectively, these tools utilize diverse methodologies to analyze and interpret the structure and behavior of binary code.

Over the past decade, static analysis tools have developed a significant new capability: the ability to intelligently infer what the original source code may have looked like. To achieve this, the software analyzes a function's control flow graph, identifying conditional jumps, the conditions being evaluated, and approximating expressions and logic structures. These pseudocode generators supplement the traditional approach, in which engineers would manually study a function's flowchart to mentally reconstruct its high-level design. However, as previously noted, some information is inevitably lost during compilation, meaning that the output of a pseudocode generator often represents only a rough approximation of the original source code.

At the same time, methods are used by malware authors specifically to try to throw off the static analysis process, by using additional "junk" values in the binary (Shinn 2011), hiding the control flow of the program (Mahoney 2023), and so on.

Thus, more recently, the desire is to assist in malware detection through new methods that can be more easily automated.

4. Current Trends in Malware Detection

Many recent research papers on malware detection only mention the DOS stub in passing. It is not considered in terms of examining the program for malware. For example, Puranik (2019) only mentions the stub, and the code within the stub would only be examined when constructing a byte-entropy histogram over the entire file. In fact, the flow diagram in this thesis indicates that all of the processing happens based on the PE header information (the COFF portion of the file). Similarly, work by Salas (2023) would examine the stub code when constructing N-grams (unigram, bigram, and so on, combinations of bytes in the raw file image). However, most

of the classification and machine learning deals with items such as the frequency of various instruction types within the executable portion of the PE file. Since this is static analysis, a call to a procedure in the stub would go undetected.

Joyce et. al. (2019) proposed a new metadata hash called Rich PE to address the issue of malware attribution. Rich PE utilizes Microsoft Rich header and the PE header metadata to improve the classification of malware as well as attribution issues stemming from techniques like polymorphism, false flags, and the use of commodity malware. The Rich header includes compiler and linker details and provides information about the build environment of a PE file. Popular metadata hashes such as the Imphash and Pehash fail to be effective against packed malware. RichPE combines Rich header metadata includes ProdIDs, compiler versions (mCVs), and counts of compilation artifacts, and applies transformations like bitLen to improve clustering capacity while maintaining low false positive rates, making it a reliable source for analysis. The authors also recognize future work as exploring spoofing methods on the PE header but do not highlight the stub vulnerabilities issue.

Author Kim (2018) investigated PE header metadata to extract relevant features from the entire header to determine if a file is malware. The project aimed to address the increasing use of obfuscation methods by malware developers to evade detection using traditional signature-based detection methods. Kim used various machine-learning techniques such as Support Vector Machines (SVM), Random Forest, K-Nearest Neighbors (K-NN), and Neural Networks to improve malware detection and classification. The paper describes the feature-free and feature-full methods for the PE header and compares their effectiveness against malware datasets from various sources. However, it does not outline any mention of the stub vulnerability explicitly in the classified malware detected in the experiment.

A similar approach is taken by Maleki et. al. (2019) in their research. After identifying the limitations of traditional malware detection techniques used by commercial antivirus programs, the authors rely on features extracted from the PE header and section table of PE files. They argue that since detection of packed malware if often difficult and inaccurate, malware must first be unpacked and then 8 selected features are applied as static features and utilized as input to machine learning algorithms for malware classification. However, the features selected for detection accuracy lack the inclusion of the stub and the potential of overlooking malware embedded in the same.

Several researchers have focused on improving the accuracy and detection time for malware analysis. Belaoued and Mazouzi (2015) developed a real-time Portable Executable malware detection system which utilized the information stored in the PEF (PE Optional Header) fields. The feature selection method included a combination of the Chi-square (KHI2) score and the Phi (j) coefficient. The authors did not include any features other than the Address of Entry point, Image Base, Section Alignment, File Alignment, Size of Headers, and Data Directory as their input from the Optional Header section.

In a similar approach, authors Zakeri et. al. (Zakeri 2015) proposed a combination of static and dynamic analysis of packed malware files. They focused on some common features of the PE file headers such as, DOS header, COFF header, PE Optional Header, data directories, import table, sections and resources related information. The rest of the features used were calculated based on certain anomalies found in these PE File headers. The input files were first parsed via static detection techniques and items that are deemed almost benign are dropped from the list. The items deemed to contain some anomaly signs are kept in the list and then parsed though dynamic detection mode. Although this. Method was successful in identifying several unknown malicious codes, the MS Dos stub was again not included in the analysis.

From the literature study so far, it is clear that both traditional and modern ML-based malware detectors concentrate on PE-file characteristics while largely neglecting the DOS header. Prior work has examined injecting adversarial perturbation bytes into various PE regions to evade ML classifiers (Aryal 2023, Aryal 2024; Aryal 2025), and only one study has touched on inserting bytes into the DOS area (Aryal 2024). That study, however, limited itself to non-functional perturbation placed in the MS-DOS stub and did not consider the risk posed by embedding the executable code, the specific threat this work investigates.

5. Crafting the Malware

To demonstrate the ability to create and hide malware in the DOS stub, the authors created a small reverse shell function, which is then embedded in the DOS header. As opposed to connecting up to a Windows machine over the network, a reverse shell is – possibly as the name implies – a program which reaches out from the Windows machine to a remote host. The reverse stub establishes a connection to the remote host where the malware

author is waiting; when they receive the connection, they are handed a command-line prompt from – in this case – Powershell.

In the following sections we describe the overall process of crafting the DOS stub code.

5.1 Writing The Code

Authoring the functionality for a reverse shell, or malware in general, is not as simple as it sounds, for two reasons:

- The function needs to be stand alone; by this we mean that the program can not directly rely on Windows libraries (DLLs) to be in memory. Rather, the program must have a way to ask the operating system to load a specific DLL and to search for functions within that DLL. A program which uses them directly will not work since there is no list of imports in the DOS header. Also, if the list of imports is present it is a potential giveaway to the malware reverse engineer.
- While the function needs to be able to look up Windows calls by name, the names cannot be ordinary strings. A string such as "LoadLibraryA", when compiled by the C/C++ compiler, would normally be placed into the ".rodata" section of the PE file, as these strings are read-only. But the code is in the DOS stub, not the PE file, so there are no sections. Rather, the string representation must be embedded within the executable bytes of the function.

To solve the second issue, in 64-bit mode an integer can occupy eight bytes. By putting two integers adjacent to each other, say an array of two eight-byte values, one can select two appropriate integer values and form a string up to 16-bytes. For example, the ASCII characters (in hexadecimal) for "LoadLibraryA" are 4C 6F 61 64 4C 69 62 72 in the first integer and 61 72 79 41 in the second. By assigning integer constants with these values, the combined array contains the string, without the compiler wanting to put it in a different section. Also note that to compare strings, one is really comparing 8-byte integers and not strings.

If we can use this method to solve the second problem it assists with us solving the first problem. On Windows, all programs have a Program Environment Block (PEB) which is visible to the program and therefore can be examined. The method to obtain the addresses of necessary library code is to first access the PEB, then use this to obtain the "program loader data" structure. This data is a collection of "loader data table entries", each of which has the name of the function. A two-step process is used, where the malware first locates the function "GetProcAddress", and then uses this to search for "LoadLibraryA" and the other functions which will be necessary for the reverse shell.

5.2 Modifying the Stub

Once the malware function is tested, we need to insert it into the DOS stub so that it will be built into a regular Windows desktop application. The overall flow for creating the hand-crafted stub code is as follows:

1. The reverse shell is written (in this case with Visual Studio and C++) using the above methods. A very small "main" program calls the reverse shell function. This also enables one to test the reverse shell before embedding it in the final file.
2. In Visual Studio, use the debugger to run to the beginning of the reverse shell function, and using the debugger in assembly language mode, note the starting and ending address of the instructions for the function.
3. While in Visual Studio, display the program memory in a hex dump window, from the start to end of the function; copy/paste this into a text editor and save it.
4. Keeping in mind that traditional malware detection relies on known patterns of bytes, the next step is to encrypt the saved hex dump so that strings (integers!) such as "LoadLibr" and "aryA" are not visible. A short C program encrypts the bytes and also prints the data out in a format which can then be included in an assembly language input file.
5. Place this result into the bottom of the DOS stub code and assemble it using the old DOS assembler. Examine the resulting MS-DOS format ".EXE" to verify the function starting point in the stub. This is the address that the Windows program will call in order to activate the malware.

The process is a rather odd mix of new technology, 64-bit instructions, and ancient voodoo techniques involving the Microsoft Macro Assembler from 1997.

One saving grace in this embedding is that since we are writing our example malware in a 64-bit environment, the 64-bit intel/AMD architecture makes the process simpler. In 64-bit mode, programs can be position-

independent – compiled so that they can be run at any location. All the data and code references are "relative" addresses as opposed to "absolute" addresses. This means that the location of our example malware in the overall PE file does not matter. If we know where in memory the program will be loaded (from the "optional" header) and we know where the malware function resides in the DOS stub (from us having placed it there), simply calling to that calculated address is all that is necessary. Additionally, although we have not yet looked into this, the aspect of Address Space Layout Randomization (ASLR) should not impact the effort since the malware code can be called based on an offset from the current location as opposed to a specific address.

5.3 Tying it all Together

The resulting artifact is a functional desktop application which brings up a window with menus, and so on. If our aim is to start up a reverse shell without the user noticing, this has to go. When the malware reverse shell is called, it reaches out to a known Internet IP address and port and establishes a connection, then starts Powershell to communicate with the remote malicious actor. Having a window pop up in front of the user, unexpectantly, is an issue. But this is as simple as just removing the windows functionality from the program, so that the only purpose of executing the program is to launch the shell code. Additionally, having a blue Powershell window appear, with no prompts or anything (the prompt was sent to the previously mentioned malicious actor) is also suspicious. But no problem, when we launch the Powershell using the Windows "CreateProcess" function, one can specify that no window is to be created for the process.

Advanced malware will do more than this. For example, a popular item for a malware author to write is code that registers as a service, so that it disappears off the list of running programs. Likewise, our reverse shell could be run as a separate thread within the program, so that it can sit quietly until the connection is established. Simply calling "CreateThread" is all that is required, and this has been tested.

6. Conclusions

Malware analysis gives defenders a clear view of how hostile software behaves and how it can bend a system to its will. By combining insights from static inspection, analysts can trace a sample's hidden paths and expose the tricks it uses to blend in, break out, or cause damage. However, these methods, at least when discussing Windows executables, generally examine only that portion of the PE file which contains the instructions pertinent to the Windows program. A combination of current technology (e.g. Visual Studio) and old technology (the Microsoft Assembler, version 6.15 and circa 2000) allows one to craft malware and place it into the file in such a way that most malware analysis tools will ignore the area where it resides. Analysis tools need to be modified to examine the entire program file, including areas which are normally ignored.

Ethics Declaration: We declare that we do not need ethical clearance for the research referred to in this paper.

AI Declaration: We declare that no AI tool has been actively used in the generation or writing of this paper.

References

Aryal, Kshitiz, Maanak Gupta, and Mahmoud Abdelsalam, (2023), "Exploiting Windows PE Structure for Adversarial Malware Evasion Attacks." Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy.

Aryal, Kshitiz, Maanak Gupta, Mahmoud Abdelsalam, Pradip Kunwar, and Bhavani Thuraisingham, (2024), "A Survey on Adversarial Attacks for Malware Analysis." IEEE Access.

Aryal, Kshitiz, Maanak Gupta, Mahmoud Abdelsalam, and Moustafa Saleh, (2024), "Explainability guided adversarial evasion attacks on malware detectors." In 2024 33rd ICCCN, pp. 1-9. IEEE.

Aryal, Kshitiz, Maanak Gupta, Mahmoud Abdelsalam, and Moustafa Saleh, (2025), "Intra-Section Code Cave Injection for Adversarial Evasion Attacks in Windows PE Malware File." Computers & Security 2025: 104690.

Belaoued, Mohamed, Smaine Mazouzi, (2015), "A Real-Time PE-Malware Detection System Based on CHI-Square Test and PE-File Features", Computer Science and its Applications, CIAA 2015 pp 416-425, available at https://link.springer.com/chapter/10.1007/978-3-319-19578-0_34

Binary Ninja, Available: https://binary.ninja [Accessed April 2025].

Defuse, Available: https://defuse.ca/online-x86-assembler.htm [Accessed April 2025].

Ghidra, Available: https://ghidra-sre.org [Accessed April 2025].

HexRays IdaPro, Available: http://www.hex-rays.com/products/ida/index.shtml [Accessed November 2024].

Hopper, Available: https://www.hopperapp.com [Accessed April 2025].

Joyce, R. J., Bilzer, K., & Burke, S. (2019). "Malware attribution using the rich header", https://raw.githubusercontent.com/RichHeaderResearch/RichPE/master/Malware%20Attribution%20Using%20the%20Rich%20Header.pdf [Accessed November 2025]

Kim, Samuel, (2018), "PE Header Analysis for Malware Detection", Master's Project, San Jose State University, available at https://scholarworks.sjsu.edu/etd_projects/624/

LinuxDevCenter, Available: http://www.linuxdevcenter.com/cmd/cmd.csp?path=o/objdump [Accessed April 2025].

Mahoney, William, J. Todd McDonald, George Grispos, Sayonnha Mandal, (2023), "Improvements on Hiding x86-64 Instructions by Interleaving", 18th International Conference on Cyber Warfare and Security, March 9-10, 2023, Towson University, Towson, Maryland.

Mahoney, W., McDonald, J. T., "Enumerating x86-64 – It's Not as Easy as Counting", Available: https://www.unomaha.edu/college-of-information-science-and-technology/research-labs/_files/enumerating-x86-64-instructions.pdf [Accessed April 2025].

Maleki, Nahid, Mehdi Bateni, Hamid Rastegari, (2019), An Improved Method for Packed Malware Detection using PE Header and Section Table Information", I. J. Computer Network and Information Security, 2019, 9, 9-17, available at https://www.mecs-press.org/ijcnis/ijcnis-v11-n9/IJCNIS-V11-N9-2.pdf

Microsoft, Available: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format [Accessed October 2025].

OllyDbg, Available: http://www.ollydbg.de/ [Accessed November 2024].

Pearce, W., "Reverse Engineering Code with IDA Pro", Elsevier Science, 2008.

Puranik, Piyushaniruddha (2019), "Static Malware Detection Using Deep Neural Networks on Portable Executables", Available: https://oasis.library.unlv.edu/cgi/viewcontent.cgi?article=4747&context=thesesdissertations Accessed October, 2025.

Relyze, Available: https://www.relyze.com/overview.html [Accessed April 2025].

Salas, M.I., & Geus, P.L. (2023). Static Analysis for Malware Classification Using Machine and Deep Learning. *2023 XLIX Latin American Computer Conference (CLEI)*, 1-10.

Shinn, S. and Mahoney, W., "Optimal Values for Disrupting x86-64 Reverse Assemblers", International Journal of Computer Science and Network Security, Volume 11, Number 11, 2011.

Udis86, Available: http://udis86.sourceforge.net/ [Accessed April 2025].

x64dbg, Available: https://x64dbg.com [Accessed April 2025].

Zakeri, Mohaddeseh, Fatemeh Faraji Daneshgar, Maghsoud Abbaspour (2015) "A static heuristic approach to detecting malware targets", Security and Communication Networks, April 8, 2015, Wiley, available at https://doi.org/10.1002/sec.1228