

A Toolkit for Detecting and Analyzing Malicious Software

Michael Weber, Matthew Schmid, David Geyer & Michael Schatz
Cigital, Inc.
21351 Ridgetop Circle
Dulles, VA 20166
{mweber, mschmid, dgeyer, mschatz}@cigital.com

Abstract

In this paper we present PEAT: The Portable Executable Analysis Toolkit. It is a software prototype designed to provide a selection of tools that an analyst may use in order to examine structural aspects of a Windows Portable Executable (PE) file, with the goal of determining whether malicious code has been inserted into an application after compilation. These tools rely on structural features of executables that are likely to indicate the presence of inserted malicious code. The underlying premise is that typical application programs are compiled into one binary, homogeneous from beginning to end with respect to certain structural features; any disruption of this homogeneity is a strong indicator that the binary has been tampered with. For example, it could now harbor a virus or a Trojan horse program. We present our investigation into structural feature analysis, the development of these ideas into the PEAT prototype, and results that illustrate PEAT's practical effectiveness.

1. Introduction

Malicious software remains a major threat to today's information systems. Detecting and analyzing dangerous programs is a costly and often inaccurate endeavor. The difficulty of this task is underscored by a recent contest challenging participants to figure out the nefarious behavior of a particular program that has already been determined to be malicious in nature [10]. Often identifying a program (or portion thereof) as malicious is half the battle. In this paper we introduce a prototype tool to aid in the analysis of potentially malicious software.

At the current stage of our work we are focusing on the detection of malicious software (malware) that has been attached to an otherwise benign host application. This is the *modus operandi* for many of the most common forms of malware including executable viruses and many Trojan

horse programs. The host program provides cover while the virus or Trojan horse performs malicious actions unbeknownst to the user. These programs often propagate while attached to games or other enticing executables.

Malicious programmers have demonstrated their creativity by developing a great number of techniques through which malware can be attached to a benign host. Several insertion methods are common, including appending new sections to an executable, appending the malicious code to the last section of the host, or finding an unused region of bytes within the host and writing the malicious content there. A less elegant but effective insertion method is to simply overwrite parts of the host application.

Given the myriad ways malicious software can attach to a benign host it is often a time-consuming process to even locate the point of infection. Traditional tools including disassemblers and debuggers may be useful for examining malware once it has been located, but provide little help in guiding an analyst to the points of interest. Malicious software hiding in a data section or other unexpected location may be particularly difficult to identify. To make matters worse, the total code size of a malicious program is frequently orders of magnitude smaller than the host that it infects.

To help a malicious software analyst quickly and efficiently locate malware within a host application we developed the Portable Executable Analysis Toolkit (PEAT). PEAT's goal is to provide methods of examining Microsoft Windows Portable Executable (PE) files [9] for signs of malicious code. We accomplish this task by developing analysis techniques that identify regions of the program that were unlikely to have been part of the host application when it was originally compiled and built. The presence of such regions is a strong indicator that malicious software has infected the host application.

This paper is organized as follows. First we summarize present technologies that address the more general problem of undesirable code (viruses, backdoor programs, etc.) in order to define the gap within that problem space that

PEAT fills. Next we describe in detail the capabilities that PEAT provides, along with the ideas behind those capabilities and their intended uses. Following that is a brief case study which illustrates our practical experience with using PEAT to detect the dangerous Trojan horse program Back Orifice 2000 [8, 5] hidden within a seemingly harmless program. A section summarizing some of PEAT's weaknesses is included. We conclude with a section describing our intentions for further improving PEAT in order to increase its effectiveness in analyzing malicious software.

2. Background

In the general case, malicious software detection is theoretically infeasible. In the specific case of searching for a particular malicious code instance, it is not only possible, but performed daily by anti-virus software. Thus, we have good commercial solutions to detecting known malicious code instances. However, the problem of determining whether software has malicious functionality is undecidable in the general case [11]. That is, we cannot look at a given application and, in general, decide whether it contains code that will result in malicious behavior. This is equivalent to the halting problem in computer science theory, which states that there is no general-purpose algorithm that can determine the behavior of an arbitrary program [7].

Aside from the halting problem, there is the definition of maliciousness to consider. What is malicious depends to a large extent on the beholder and the context. For example, a disk formatting program might be exactly what the user wants (and therefore is not considered malicious), though when embedded in a screensaver unbeknownst to the user, it can be considered malicious. Thus, we cannot develop an algorithm to decide maliciousness.

Other seminal work in this area has proved the undecidability properties of detecting malicious software in the general case in different contexts [3, 4, 12]. So, while the prior art has demonstrated that detecting malicious code in the general case is undecidable, what options are we left with in addressing the unknown malicious code problem?

One approach to detecting malicious code in executable programs is being investigated by the LFSM Research Group [2] in which both static and dynamic methods are applied in order to perform model checking to ensure that the program under analysis will not violate any stated security policies. This is an example of an approach to malicious software detection that requires an analyst to define malicious behavior in the form of a policy.

Our present research differs from traditional approaches to the malicious code problem in that it does not attempt to define or identify malicious *behavior*. Instead, the research focuses on *structural* characteristics of malicious executable code. This approach allows for methods of examining any application, whether previously known or un-

known, in order to determine if it has been tampered with since its original development. Such tampering usually takes the form of an embedded virus or Trojan horse that is activated during subsequent executions of the executable program.

We chose Microsoft Windows as the initial platform for PEAT because of its market dominance and the proliferation of malicious code targeted at that operating system and its applications. As other platforms increase in popularity, malicious attacks on those platforms are sure to follow. We designed PEAT with this in mind so that it may be easily extended to accommodate other executable file formats, such as ELF.

3. PEAT: Portable Executable Analysis Toolkit

The Portable Executable Analysis Toolkit (PEAT) provides an analyst with an array of tools for examining Windows Portable Executable (PE) files for signs of malicious code. Future work will extend PEAT to include features to help understand the capabilities of that software. These tools are designed to locate structural features of executables that do not fit in with surrounding regions; i.e., regions of bytes that appear to have been inserted into an otherwise homogeneous binary file. The underlying premise is that programs are typically compiled into one consistent binary. Any deviation from this self-consistency is a strong indicator of tampering. The program may be infected with a virus, it could contain a Trojan horse program, or it could have been modified in some other manner resulting in a program whose behavior is different from the intended behavior of the original program.

3.1. Overview

PEAT's tools fall into three general categories: simple static checks, visualization, and automated statistical analysis. Simple static checks consist of a list of features whose presence or absence PEAT attempts to verify in order to quickly gain information that might suggest something suspicious. For example, PEAT immediately issues a warning if the program's entry point is in an unusual location. Visualization tools include graphical depictions of several features of the PE file. Examples of these include

- probabilities that regions of bytes in the PE file contain code, padding, ASCII data, or random byte values
- address offsets for instructions that perform operations such as jumps, calls, or register accesses
- patterns of instructions that are known to indicate certain behavior (e.g., pushing arguments onto the stack and making a call)

The visualization toolkit also uses PEAT's disassembler, based on work by Watanabe [13], to parse and decode instructions. The user may then view the disassembly listing. In addition, in order to identify ASCII strings, the user may view the ASCII representation of all byte values within a given region. Together, these visualization tools are intended to allow an expert analyst to explore an executable file in an attempt to identify regions that appear to be inconsistent with the entire program. To complement these manual analysis capabilities, PEAT also provides automated analysis tools to guide the analyst to suspicious regions of the PE file.

PEAT's automated analysis tools perform statistical tests in order to detect anomalous regions. The analysis operates on many of the same PE file characteristics as the visualization tools. The user chooses which features to consider, and the analysis engine will then divide the PE file sections into several regions and determine whether there are any statistically significant differences between those regions. Each anomaly that is found is reported and stored as a suspect region, along with automatically generated comments describing why it stands out.

The remainder of this section describes these tools in detail.

3.2. Static Checks

PEAT performs several static checks of the PE file under analysis to quickly gain information that might suggest that it contains something suspicious. The first of these is a check on the program's entry point address, obtained from the PE file header. This address should fall within some section that is marked as executable (typically this will be the first section, and named .text or CODE). If this is not the case, for example, if the entry point lies in the .reloc section, which should not contain executable code, a warning will be displayed in PEAT's main window once the PE file is loaded into PEAT.

Another static check attempts to identify "bogus calls", which we define to be instructions that call to the immediately following instruction. Such a sequence of instructions is a common method viruses use to determine their address in memory. This is because the value of the instruction pointer register EIP gets pushed onto the stack as a side effect of a CALL instruction [1]. The virus exploits this by immediately popping this value. Because of the suspicious nature of such instruction sequences, PEAT will alert the user to their presence after the PE file has been disassembled.

Finally, PEAT determines which DLL libraries are listed in the PE file's import table and reports the name and contents of each. In addition, it finds all instructions in the program that call a function in a DLL library and reports the locations of those instructions, along with the library and

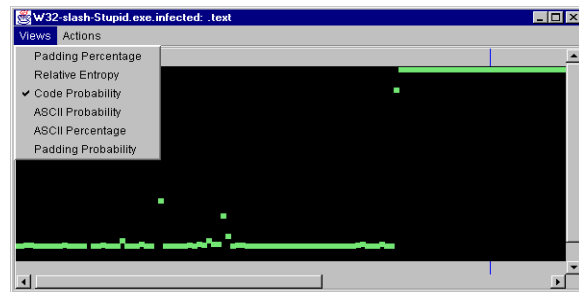


Figure 1. Byte-type section view: code likelihood

function name. This is a quick initial pass at determining whether the program has any unexpected capabilities, such as file or network I/O in an application that should not require that functionality.

3.3. Visualization Tools

Byte-Type Views: The visualization toolkit provides multiple ways to view structural features of a PE file. One such view is a plot that allows an analyst to quickly see which regions of a PE file contain code, ASCII data, padding for alignment, or random byte values. An example is given in Figure 1 in which the .text segment of the W32/Stupid virus is displayed. Each point along the horizontal axis represents a window of bytes of the .text segment, and its value along the vertical axis, scaled from 0 to 1, represents how likely the window of bytes consists of some byte-type of interest, with higher values indicating greater likelihood. In this example, code likelihood is displayed. What we see is that only the latter portion of this .text segment appears to contain real code, indicated by the fairly solid line of points high on the vertical axis.

The probability values are determined by standard statistical proportion tests in which the proportion of a certain set of byte values (e.g., values in the ASCII character range) observed in a window of bytes is computed. Based on the size of the window, the size of the set of target byte values, and the observed proportion of those target values, the probability p of drawing the observed byte values from a population of random byte values is computed.¹ The complement of p is plotted, so that higher values indicate greater likelihood that the window contains the byte-type of interest. In addition to these probabilities, the observed proportion of each of these byte-types is also available for viewing.

¹Here, $p = 1 - F(z)$ where F is the cumulative distribution function of a standard normal random variable and $z = \frac{x - n * e}{\sqrt{n * e * (1 - e)}}$ with n being the window size, e being the expected proportion, and x/n being the observed proportion.

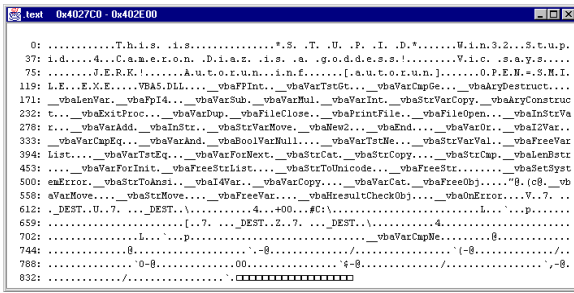


Figure 2. ASCII view

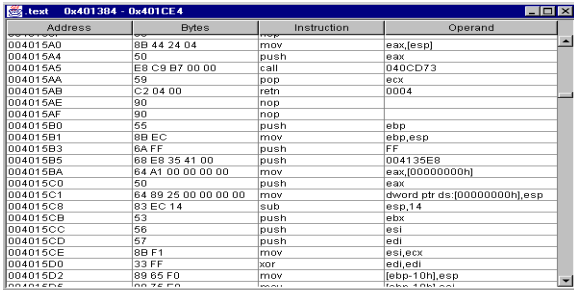


Figure 3. Disassembly view

The other type of information that is available in this view is a plot of byte-value entropy. That is, the section is divided into several windows and the total entropy of the byte values in that window is computed. These entropy values are then normalized against the total entropy values for each window and then plotted on the vertical axis.

ASCII View: From the section-level view described above, the user may select a region to investigate further. One additional view is a display of the ASCII representation of each byte in the selected region. An example is displayed in Figure 2. These bytes correspond to the region from Figure 1 that has a high probability of containing ASCII data.

Disassembly View: Another way to investigate a particular region of interest is to have PEAT disassemble that region and display the results. For each instruction that is parsed, the address, raw byte values, instruction name, and the operands of the instruction are displayed. An example is shown in Figure 3.

Memory Access via Register Offsets: PEAT provides a view that allows the user to see whenever memory is accessed by adding an offset to a register value in order to determine an address in memory. The user first chooses a register to consider, such as the base pointer register EBP, and then PEAT uses the disassembly information to find and plot all such memory accesses. An example is shown below. There is a horizontal line through the middle of the vertical axis representing 0, and positive and negative offsets are

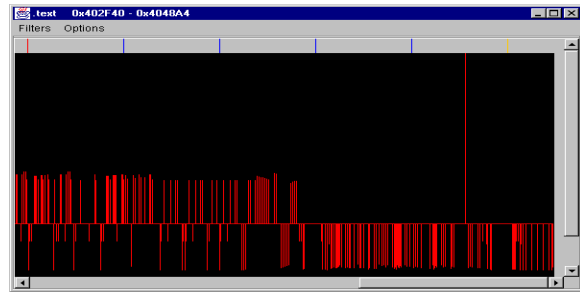


Figure 4. Register offset view

plotted against this. This view can be used to visually assess whether some region uses this means of accessing memory differently from other regions (e.g., larger offsets, more frequent offsets, or offsets in opposite directions).

Other Views: We have defined some other views that have not yet been incorporated into PEAT. One such view displays offsets for jump and call instructions, similar to the view of register offsets, in order to visually determine whether regions are fairly self-contained or whether large jumps are made, such as to outside the boundaries of the executing segment. Another view displays common instruction patterns, such as several pushes followed by call (indicating the pushing of arguments onto the stack in order to make a procedure call). The analyst could visually note the presence or absence of these common patterns and determine whether any region of the section appears to be different from the others.

3.4. Statistical Analysis

As a supplement to the simple static checks and the visualization tools, PEAT also provides analysis capabilities based on using statistical methods for identifying anomalous regions within a PE file. The user may choose from a wide range of features to extract from the program, such as:

- Instruction frequencies
- Instruction patterns
- Register offsets
- Jump and Call offsets
- Entropy of opcode values
- Code and ASCII probabilities

These are discussed in further detail below, in terms of their potential usefulness in identifying anomalous regions within a program. But first we discuss the general statistical approach that is applied for whichever features are used as input data.

When PEAT performs its automated analysis, it iterates over each section of the PE file. The section is disassembled into instructions, and then divided into n consecutive disjoint windows of a fixed number of instructions. The metric of interest for each window is computed (e.g., entropy of opcode values), yielding a list of values

$$X = (x_1, x_2, \dots, x_n)$$

From this list, another list of differences

$$Y = (y_1, y_2, \dots, y_{n-1})$$

is computed, where $y_i = x_{i+1} - x_i$.

Next, PEAT iterates over the windows and determines for each window whether the corresponding data point in X is a statistical outlier with respect to the remaining data points in X . For window i , the mean and standard deviation of $X \setminus x_i$ is computed, and it is determined whether x_i lies within two standard deviations of the mean. Anytime this is not the case, the window will be reported as anomalous, along with a probability reflecting the likelihood of realizing a value at least as deviant as x_i from the remaining empirical distribution. This procedure yields a list of windows that have, for example, anomalous entropy, with respect to the other windows in the section.

A similar procedure is applied to the windows with respect to the Y data points, yielding a list of windows that exhibit a significant sequential change in the metric of interest. For example, if common instruction patterns have been observed up to some point in the section, and then all of a sudden disappear, this will be reported. The reasoning behind using both the X and Y points is that the X points may be insufficient to find an anomalous region in a section whose first half, for example, is normal, while its entire second half has been overwritten with malicious code.

Given this general framework for statistical analysis, PEAT provides several different metrics from which to build a set of criteria for anomaly detection.

Instruction Frequencies: The idea behind examining instruction frequencies from window to window stems from one of our more fundamental premises that viruses tend to be written in assembly language while the host applications tend to be compiled from high-level languages. We performed a study based on this premise to identify any instructions that appear frequently in assembly language programs and rarely in compiled code, and similarly, instructions that appear frequently in compiled code and rarely in assembly language. The results of this study led to the lists of instructions whose frequencies are calculated for the purpose of finding anomalous windows. Ideally, malicious assembled code that has been injected into a section of a PE file will be discovered during the statistical analysis due to a sudden absence of frequent compiled code instructions, and further

analysis could verify that assembly language instructions are abnormally frequent in that region.

Instruction Patterns: The motivation for examining patterns of instructions is very similar to the ideas behind examining instruction frequencies. Our premise is that compiled code is likely to exhibit regular instruction sequences to implement common constructs like function calls and returns and looping constructs. An assembly language programmer's conventions for implementing these are not necessarily the same as the compiler, and perhaps not even consistent from use to use. We have performed an initial study of assembly language output from the Microsoft Visual C++ compiler and have built a list of patterns that are seen to result from the use of common high-level language constructs. The frequencies of the patterns are one metric that the user can choose to incorporate into an analysis with the goal being to discover injected malicious assembly language code via the sudden absence of such patterns.

Memory Access via Register Offsets: Another premise we have is that normal applications and malicious code will each use certain registers differently. In particular, the base pointer register EBP is commonly used by normal applications as a reference point for accessing local variables on the stack. Malicious programs, however, can take advantage of this key reference point to determine where they are in memory, a commonly necessary piece of information for them to function and adjust as they spread throughout unknown executables. Thus register offset values used when accessing memory via a register are another metric that can be used during statistical analysis.

Jump and Call Distances: The common layout of an application compiled from a high-level language is simply a sequence of self-contained functions. Control flows between these functions via the CALL and RET instructions. Jump instructions alter the control flow within a single function, implementing high level conditional constructs such as `if` statements and `while` loops. Therefore, the distances traveled during a normal application's jump instructions should be relatively small and regular, and similarly, the distances traveled during call instructions should be relatively larger and regular. What should very rarely be observed in normal applications are extremely large jump or call distances, such as to other sections of the PE file.

Byte-Type Probabilities: The last types of information that PEAT uses as input to the statistical analysis are the probabilities that windows consist of ASCII data, padding, or real code. This is the same information that is presented in the section view display. In conjunction with the other metrics, this byte-type information can aid in the further investigation of regions that are marked as anomalous. For example, if a window is marked as an outlier for having a sudden absence of common instruction patterns, but it is

also marked as an outlier for having a sudden high probability of being padding and low probability of being code, the analyst can more confidently conclude that the absence of patterns does not indicate the presence of assembly language code but rather the absence of code altogether.

When the entire automated analysis completes, the analyst is presented with a list of windows that were found to be anomalous. Each is reported along with its location in the section and a description of what characteristics made it stand out. From this list, the analyst can easily invoke the visualization options, such as the disassembly, in order to further investigate some particular region.

4. Results

We have had initial success with using Peat to perform analysis on several malicious code samples. Of particular interest is a study in which we detected the Back Orifice 2000 server [8, 5] hiding inside of a seemingly harmless application.

InPEct is a tool that is used to inject arbitrary Trojans into arbitrary victim applications on the Windows platform. When a user runs the resulting executable, the injected Trojan will start to run in the background, and the victim application will run as usual. The Trojan persists after the victim application terminates. We used InPEct to inject the BO2K server into a typical Windows application: the calculator program, calc.exe.

We then examined the infected executable with Peat in order to determine if any of Peat's metrics could detect the Trojan's presence and give insight into the infection method. The process of that examination is presented here, along with the results.

When Peat first loads a PE file, it displays several pieces of information from the PE header, including a list of the file's sections and its entry point. Upon loading the infected calculator program, Peat issues a warning that the program's entry point is in an unusual place. The program control begins in the .rsrc section, as opposed to the expected .text section. In addition, we see that the .rsrc section is unusually long, compared to that of typical applications. This information, shown in Figure 5, is the first piece of evidence that the original application has been tampered with.

Next, Peat performs its automated analysis in order to identify anomalous regions within each section of the file. It identifies two anomalous windows in the .rsrc section and displays these as shown in Figure 6. The first window is an outlier with respect to the code probability metric. It has a high likelihood of containing code, but the other windows in this section do not. An analyst should recognize this as quite suspicious, as the .rsrc section does not typically contain code. However, this finding is consistent with Peat's entry point warning. The entry point happens to lie in this anomalous window.

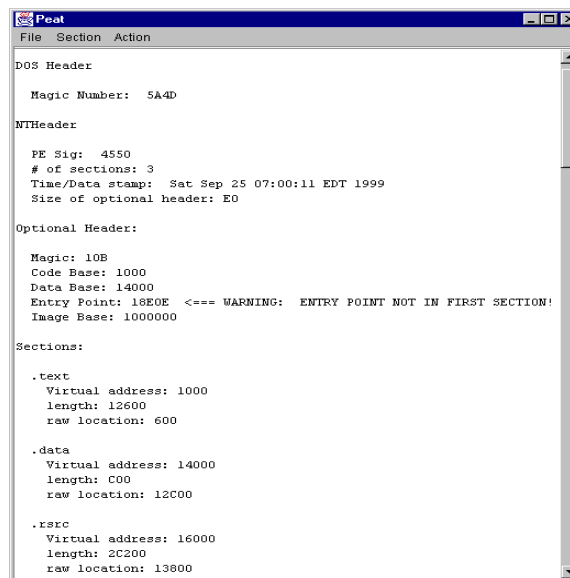


Figure 5. PE header information for the infected calculator program

This window also stands apart from the remainder of the section due to an abnormal level of entropy. The second anomalous window was marked as having an unusual change in entropy, with respect to the previous window. This pair of warnings (an abnormal value, followed by a drastic change in that value, for any given metric) typically suggests that something has been inserted into the original application. To investigate these entropy levels further, the analyst uses the visualization tools to examine the .rsrc section.

The view of the entropy metric for the .rsrc section is shown in Figure 7.² We observe that there is a point at which the entropy level drastically increases. This is a strong indicator that an encrypted region of bytes is present.

At this point, the analyst has gathered sufficient evidence to recognize a particular pattern. A common technique that Trojan injection tools use, including InPEct, is to encrypt and append the Trojan to the end of the last section of the victim application. The injection method also inserts some additional code to tend to matters such as decrypting the Trojan, running it, and returning control to the original application. Finally, it changes the program entry point to this newly inserted startup routine.

After observing all of this evidence, the analyst may confidently conclude that the file under investigation appears to have been infected with a Trojan via the method described

²Only the first portion is actually shown. The user would scroll to the right to see the remainder, which looks similar to the latter portion of the data shown here.

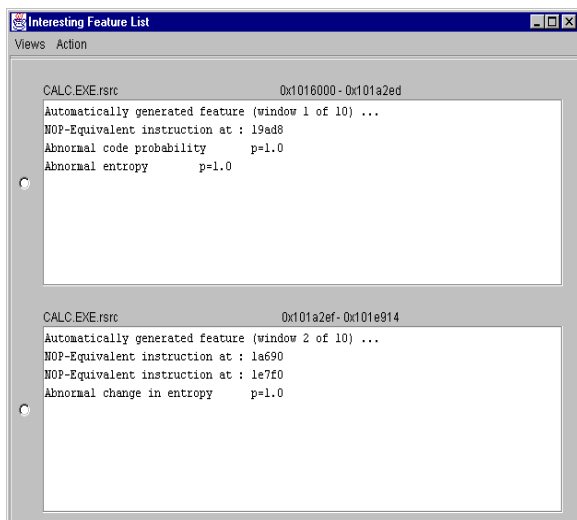


Figure 6. Two anomalous regions of the .rsrc section

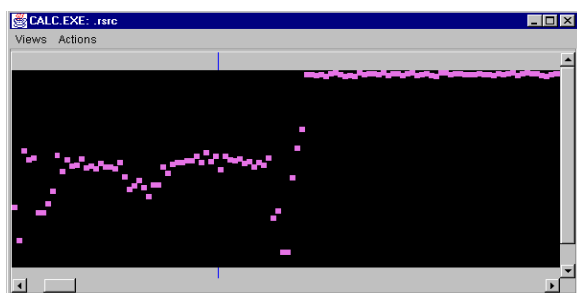


Figure 7. The entropy metric for the .rsrc section of the calculator program

above. If the analyst is interested in examining the file further, the disassembly view of the inserted startup routine is available as a useful starting point. Other features would be useful to an analyst in this scenario, such as determining the identity or capabilities of the inserted Trojan. The current implementation of Peat does not yet address these issues, but they are discussed in the following section.

5. Weaknesses

Malicious software detection technologies tend to suffer from a common problem: once an attacker knows the criteria that drive the detection logic, he can adapt his attack to circumvent detection. PEAT, to some extent, is also subject to this. Although PEAT has a collection of several independent criteria, a determined attacker could specialize an attack to avoid PEAT's detection methods. For exam-

ple, an analyst using PEAT would have difficulty detecting any of the following attacks, all of which lie within PEAT's intended scope of infected executables:

- If the host executable is completely overwritten and replaced by the malicious code, PEAT will not detect any inconsistencies during the analysis. However this attack is not very appealing to an attacker because the victim user will immediately become suspicious after running the host executable and observing that it did not run properly.
- If the attacker has some knowledge of a particular host application's source and compilation history, he could develop his malicious code in a similar fashion, so that the metrics that PEAT computes would have similar values across the malicious code and the host application. For example, he could choose to develop a Trojan in Visual C++ for the sole purpose of having that Trojan masquerade as a function belonging to the host application that was known to be developed in that language. Fortunately, this would only be the first step along the path to avoiding detection in this manner. Other factors such as compiler optimization levels and even coding styles may result in the Trojan exhibiting outstanding patterns. Further, the attacker still faces the problem of modifying the host so that control flows to the Trojan, and PEAT has proven to be effective at identifying various methods of doing this.
- It is possible to infect a host application with a very small amount of code that simply loads a separate DLL containing a malicious payload or perhaps starts another process. PEAT is currently limited in its ability to handle this attack. It can alert the analyst to the presence of calls to common DLL functions like `LoadLibrary()` and `CreateProcess()`. However it does not descend into all such libraries or separate executables in order to analyze them in the context of the main file under analysis.

In addition there are other attacks on executables that lie outside of the scope of PEAT. For example, some viruses attack executable images in memory, as opposed to the file stored on disk.

Finally, PEAT may sometimes report anomalies that do not necessarily indicate the presence of malicious code. For example, it may report that at the end of the `.text` segment, the byte value entropy suddenly and drastically changes. Further inspection might reveal that this is due to the presence of section alignment padding and not some alteration of the original file. This is not a false alarm in the traditional sense, as PEAT is not intended to be used as an automatic detection tool. However, it does reveal that an analyst requires some degree of domain knowledge about PE files,

viruses, and other system-level concepts, as well as some experience working with PEAT and learning how to interpret its output from various metrics in order to perform a sound analysis.

6. Future Work

The main accomplishment of this work was the identification of several structural aspects of Windows executables that can reliably indicate the presence of malicious code. The next major feature that we plan to incorporate into PEAT is a component that can analyze the capabilities of a region of code. PEAT already provides the identification of imported DLLs and the location of calls to DLL functions. We plan to take this a step further by using this information to determine what specific actions the code is capable of performing. In addition, we plan to incorporate the ability to recursively descend into unknown imported DLLs to determine their capabilities. For example, if the PE file imports DLL *unknown.dll*, and its function `f○○()` is called, we would like to determine the capabilities of that function as well.

Along the lines of analyzing code's capabilities, we have noted that many forms of malicious code, in particular viruses, reuse sections of code from other malicious programs. These reused sections of code can free the malicious code writer from having to rewrite complicated functionality like infection routines or encryption routines. It is desirable to be able to identify such common sequences of instructions. In fact, an analyst should be able to draw on an entire database of known, and possibly documented, malicious code building blocks so that when these are encountered during analysis, the analyst can quickly determine that code's functionality. With these ideas in mind, we plan to implement a framework for maintaining a collection of these common code patterns and incorporate this into PEAT.

Finally, the HoneyNet group is currently hosting the Reverse Challenge project in which an unknown malicious Linux executable is to be analyzed in order to determine its capabilities and origin [10]. Participants in this contest are typically using tools such as IDA Pro Disassembler [6] and Fenris [14] to examine the ELF format binary. The current implementation of PEAT serves as a useful complement to such tools. For example, the IDA disassembler is quite useful for analyzing code, but its effectiveness could be greatly augmented by coupling it with PEAT. In this arrangement, the analyst may not need to analyze the assembly of an entire program but only particular regions that PEAT identifies as being suspicious, thereby greatly reducing the time invested in the analysis. We are examining useful features related to program understanding that these two tools provide. More importantly, we are noting missing features that would be beneficial to an analyst, so that future versions of

PEAT may fill these gaps.

References

- [1] *IA-32 Intel Architecture Software Developer's Manual, Volume 2*. Intel, 2001.
- [2] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie and N. Tawbi. Static Detection of Malicious Code in Executable Programs. *Symposium on Requirements Engineering for Information Security (SREIS'01)*. March 5-6, 2001.
- [3] F. Cohen. Computer viruses. *Computers & Security*, 6(1):22–35, 1987.
- [4] F. Cohen. Computational aspects of computer viruses. *Computers & Security*, 8(4):325–344, 1989.
- [5] Cult of the Dead Cow. Back Orifice 2000 website. Available at <http://bo2k.sourceforge.net>. May, 2002.
- [6] DataRescue. IDA Pro Disassembler website. Available at <http://www.datarescue.com/idabase/>. May, 2002.
- [7] M.E. Davis and E.J. Weyuker. *Computability, Complexity, and Languages*. Academic Press, 1983.
- [8] Internet Security Systems. *ISS X-Force White Paper: Back Orifice 2000 Backdoor Program*. Available at <http://documents.iss.net/whitepapers/bo2k.pdf>. July, 1999.
- [9] M. Pietrek. *Windows 95 System Programming Secrets*. IDG Books, 1995.
- [10] Project HoneyNet. The Reverse Challenge website. Available at <http://project.honeynet.org/reverse/>. July, 1999.
- [11] A. Rubin and D. Geer. Mobile code security. *IEEE Internet Computing*, 2(6), November/December 1998.
- [12] H. Thimbleby, S. Anderson, and P. Cairns. A framework for modeling trojans and computer virus infection. *Computer Journal*, 41(7):444–458, 1999.
- [13] T. Watanabe. *How to write a disassembler*. 2000.
- [14] M. Zalewski. Fenris website. Available at <http://razor.bindview.com/tools/fenris/>. May, 2002.