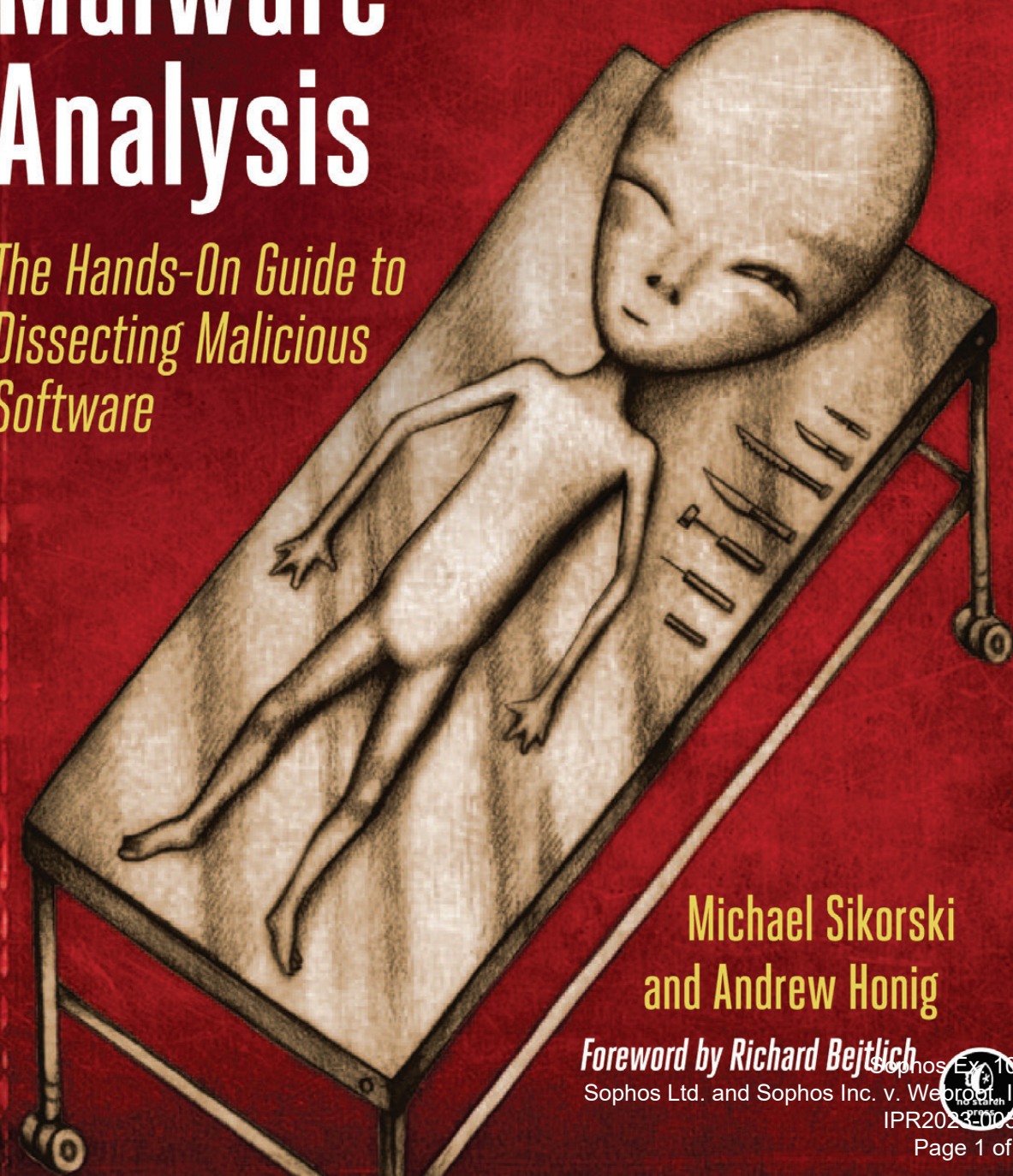


Practical Malware Analysis

*The Hands-On Guide to
Dissecting Malicious
Software*



**Michael Sikorski
and Andrew Honig**

Foreword by Richard Bejtlich

Sophos Ltd. and Sophos Inc. v. Webrobot, Inc.
IPR2023-00528

Page 1 of 12

PRAISE FOR *PRACTICAL MALWARE ANALYSIS*

“An excellent crash course in malware analysis.”

—**Dino Dai Zovi**, INDEPENDENT SECURITY CONSULTANT

“... the most comprehensive guide to analysis of malware, offering detailed coverage of all the essential skills required to understand the specific challenges presented by modern malware.”

—**Chris Eagle**, SENIOR LECTURER OF COMPUTER SCIENCE, NAVAL POSTGRADUATE SCHOOL

“A hands-on introduction to malware analysis. I'd recommend it to anyone who wants to dissect Windows malware.”

—**Ilfak Guilfanov**, CREATOR OF IDA PRO

“... a great introduction to malware analysis. All chapters contain detailed technical explanations and hands-on lab exercises to get you immediate exposure to real malware.”

—**Sebastian Porst**, GOOGLE SOFTWARE ENGINEER

“... brings reverse-engineering to readers of all skill levels. Technically rich and accessible, the labs will lead you to a deeper understanding of the art and science of reverse-engineering. I strongly recommend this book for beginners and experts alike.”

—**Danny Quist**, PHD, FOUNDER OF OFFENSIVE COMPUTING

“If you only read one malware book or are looking to break into the world of malware analysis, this is the book to get.”

—**Patrick Engbretson**, IA PROFESSOR, DAKOTA STATE UNIVERSITY AND AUTHOR OF *The Basics of Hacking and Pen Testing*

“... an excellent addition to the course materials for an advanced graduate level course on Software Security or Intrusion Detection Systems. The labs are especially useful to students in teaching the methods to reverse-engineer, analyze, and understand malicious software.”

—**Sal Stolfo**, PROFESSOR, COLUMBIA UNIVERSITY

WARNING

This is a book about malware. The links and software described in this book are *malicious*. Exercise extreme caution when executing unknown code and visiting untrusted URLs.

For hints about creating a safe virtualized environment for malware analysis, visit Chapter 2. Don't be stupid; secure your environment.

PRACTICAL MALWARE ANALYSIS

The Hands-On Guide to Dissecting Malicious Software

by Michael Sikorski and Andrew Honig



**no starch
press**

San Francisco

PRACTICAL MALWARE ANALYSIS. Copyright © 2012 by Michael Sikorski and Andrew Honig.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

16 15 14 13 12 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-290-1

ISBN-13: 978-1-59327-290-6

Publisher: William Pollock

Production Editor: Alison Law

Cover Illustration: Hugh D'Andrade

Interior Design: Octopod Studios

Developmental Editors: William Pollock and Tyler Ortman

Technical Reviewer: Stephen Lawler

Copyeditor: Marilyn Smith

Compositor: Riley Hoffman

Proofreader: Irene Barnard

Indexer: Nancy Guenther

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

38 Ringold Street, San Francisco, CA 94103

phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data

A catalog record of this book is available from the Library of Congress.

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the authors nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

Once you identify which files require full analysis, it's time to develop signatures to detect malware infections on your network. As you'll learn throughout this book, malware analysis can be used to develop host-based and network signatures.

Host-based signatures, or indicators, are used to detect malicious code on victim computers. These indicators often identify files created or modified by the malware or specific changes that it makes to the registry. Unlike antivirus signatures, malware indicators focus on what the malware does to a system, not on the characteristics of the malware itself, which makes them more effective in detecting malware that changes form or that has been deleted from the hard disk.

Network signatures are used to detect malicious code by monitoring network traffic. Network signatures can be created without malware analysis, but signatures created with the help of malware analysis are usually far more effective, offering a higher detection rate and fewer false positives.

After obtaining the signatures, the final objective is to figure out exactly how the malware works. This is often the most asked question by senior management, who want a full explanation of a major intrusion. The in-depth techniques you'll learn in this book will allow you to determine the purpose and capabilities of malicious programs.

Malware Analysis Techniques

Most often, when performing malware analysis, you'll have only the malware executable, which won't be human-readable. In order to make sense of it, you'll use a variety of tools and tricks, each revealing a small amount of information. You'll need to use a variety of tools in order to see the full picture.

There are two fundamental approaches to malware analysis: static and dynamic. *Static analysis* involves examining the malware without running it. *Dynamic analysis* involves running the malware. Both techniques are further categorized as basic or advanced.

Basic Static Analysis

Basic static analysis consists of examining the executable file without viewing the actual instructions. Basic static analysis can confirm whether a file is malicious, provide information about its functionality, and sometimes provide information that will allow you to produce simple network signatures. Basic static analysis is straightforward and can be quick, but it's largely ineffective against sophisticated malware, and it can miss important behaviors.

Basic Dynamic Analysis

Basic dynamic analysis techniques involve running the malware and observing its behavior on the system in order to remove the infection, produce effective signatures, or both. However, before you can run malware safely, you must set up an environment that will allow you to study the running

malware without risk of damage to your system or network. Like basic static analysis techniques, basic dynamic analysis techniques can be used by most people without deep programming knowledge, but they won't be effective with all malware and can miss important functionality.

Advanced Static Analysis

Advanced static analysis consists of reverse-engineering the malware's internals by loading the executable into a disassembler and looking at the program instructions in order to discover what the program does. The instructions are executed by the CPU, so advanced static analysis tells you exactly what the program does. However, advanced static analysis has a steeper learning curve than basic static analysis and requires specialized knowledge of disassembly, code constructs, and Windows operating system concepts, all of which you'll learn in this book.

Advanced Dynamic Analysis

Advanced dynamic analysis uses a debugger to examine the internal state of a running malicious executable. Advanced dynamic analysis techniques provide another way to extract detailed information from an executable. These techniques are most useful when you're trying to obtain information that is difficult to gather with the other techniques. In this book, we'll show you how to use advanced dynamic analysis together with advanced static analysis in order to completely analyze suspected malware.

Types of Malware

When performing malware analysis, you will find that you can often speed up your analysis by making educated guesses about what the malware is trying to do and then confirming those hypotheses. Of course, you'll be able to make better guesses if you know the kinds of things that malware usually does. To that end, here are the categories that most malware falls into:

Backdoor Malicious code that installs itself onto a computer to allow the attacker access. Backdoors usually let the attacker connect to the computer with little or no authentication and execute commands on the local system.

Botnet Similar to a backdoor, in that it allows the attacker access to the system, but all computers infected with the same botnet receive the same instructions from a single command-and-control server.

Downloader Malicious code that exists only to download other malicious code. Downloaders are commonly installed by attackers when they first gain access to a system. The downloader program will download and install additional malicious code.

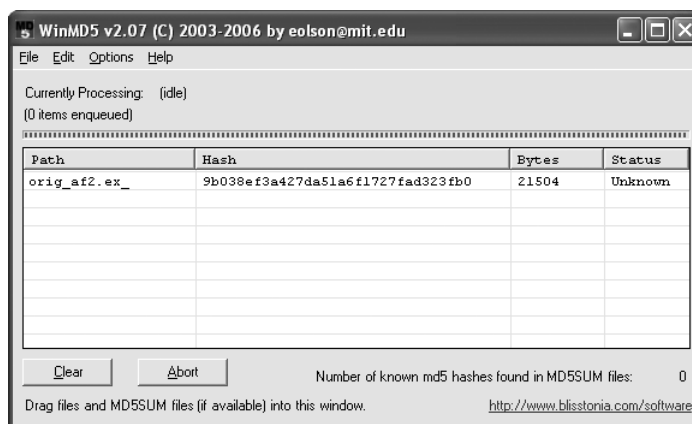


Figure 1-1: Output of WinMD5

Finding Strings

A *string* in a program is a sequence of characters such as “the.” A program contains strings if it prints a message, connects to a URL, or copies a file to a specific location.

Searching through the strings can be a simple way to get hints about the functionality of a program. For example, if the program accesses a URL, then you will see the URL accessed stored as a string in the program. You can use the Strings program (<http://bit.ly/ic4plL>), to search an executable for strings, which are typically stored in either ASCII or Unicode format.

NOTE *Microsoft uses the term wide character string to describe its implementation of Unicode strings, which varies slightly from the Unicode standards. Throughout this book, when we refer to Unicode, we are referring to the Microsoft implementation.*

Both ASCII and Unicode formats store characters in sequences that end with a *NULL terminator* to indicate that the string is complete. ASCII strings use 1 byte per character, and Unicode uses 2 bytes per character.

Figure 1-2 shows the string BAD stored as ASCII. The ASCII string is stored as the bytes 0x42, 0x41, 0x44, and 0x00, where 0x42 is the ASCII representation of a capital letter *B*, 0x41 represents the letter *A*, and so on. The 0x00 at the end is the NULL terminator.

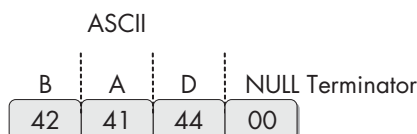


Figure 1-2: ASCII representation of the string BAD

Figure 1-3 shows the string BAD stored as Unicode. The Unicode string is stored as the bytes 0x42, 0x00, 0x41, and so on. A capital *B* is represented by the bytes 0x42 and 0x00, and the NULL terminator is two 0x00 bytes in a row.

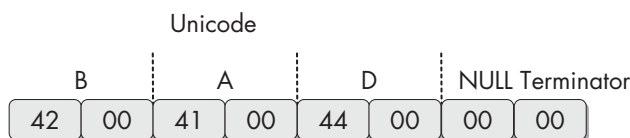


Figure 1-3: Unicode representation of the string BAD

When Strings searches an executable for ASCII and Unicode strings, it ignores context and formatting, so that it can analyze any file type and detect strings across an entire file (though this also means that it may identify bytes of characters as strings when they are not). Strings searches for a three-letter or greater sequence of ASCII and Unicode characters, followed by a string termination character.

Sometimes the strings detected by the Strings program are not actual strings. For example, if Strings finds the sequence of bytes 0x56, 0x50, 0x33, 0x00, it will interpret that as the string VP3. But those bytes may not actually represent that string; they could be a memory address, CPU instructions, or data used by the program. Strings leaves it up to the user to filter out the invalid strings.

Fortunately, most invalid strings are obvious, because they do not represent legitimate text. For example, the following excerpt shows the result of running Strings against the file *bp6.ex_*:

```
C:>strings bp6.ex_
VP3
VW3
t$@
D$4
99.124.22.1 ❶
e-@
GetLayout ❶
GDI32.DLL ❸
SetLayout ❷
M}C
Mail system DLL is invalid.!Send Mail failed to send message. ❺
```

In this example, the bold strings can be ignored. Typically, if a string is short and doesn't correspond to words, it's probably meaningless.

On the other hand, the strings GetLayout at ❶ and SetLayout at ❷ are Windows functions used by the Windows graphics library. We can easily identify these as meaningful strings because Windows function names normally begin with a capital letter and subsequent words also begin with a capital letter.

GDI32.DLL at ❸ is meaningful because it's the name of a common Windows *dynamic link library (DLL)* used by graphics programs. (DLL files contain executable code that is shared among multiple applications.)

As you might imagine, the number 99.124.22.1 at ❹ is an IP address—most likely one that the malware will use in some fashion.

Finally, at ❺, Mail system DLL is invalid.!Send Mail failed to send message. is an error message. Often, the most useful information obtained by running Strings is found in error messages. This particular message reveals two

things: The subject malware sends messages (probably through email), and it depends on a mail system DLL. This information suggests that we might want to check email logs for suspicious traffic, and that another DLL (Mail system DLL) might be associated with this particular malware. Note that the missing DLL itself is not necessarily malicious; malware often uses legitimate libraries and DLLs to further its goals.

Packed and Obfuscated Malware

Malware writers often use packing or obfuscation to make their files more difficult to detect or analyze. *Obfuscated* programs are ones whose execution the malware author has attempted to hide. *Packed* programs are a subset of obfuscated programs in which the malicious program is compressed and cannot be analyzed. Both techniques will severely limit your attempts to statically analyze the malware.

Legitimate programs almost always include many strings. Malware that is packed or obfuscated contains very few strings. If upon searching a program with Strings, you find that it has only a few strings, it is probably either obfuscated or packed, suggesting that it may be malicious. You'll likely need to throw more than static analysis at it in order to investigate further.

NOTE *Packed and obfuscated code will often include at least the functions LoadLibrary and GetProcAddress, which are used to load and gain access to additional functions.*

Packing Files

When the packed program is run, a small wrapper program also runs to decompress the packed file and then run the unpacked file, as shown in Figure 1-4. When a packed program is analyzed statically, only the small wrapper program can be dissected. (Chapter 18 discusses packing and unpacking in more detail.)

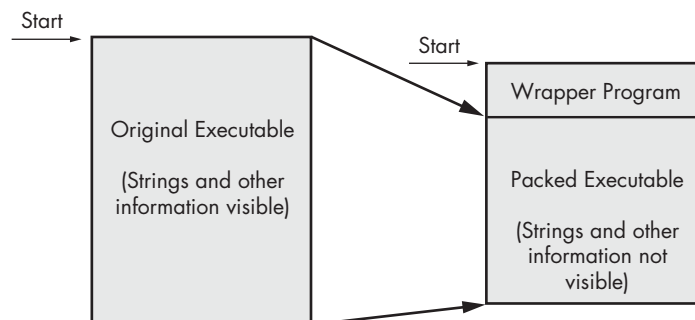


Figure 1-4: The file on the left is the original executable, with all strings, imports, and other information visible. On the right is a packed executable. All of the packed file's strings, imports, and other information are compressed and invisible to most static analysis tools.

18

PACKERS AND UNPACKING

Packing programs, known as *packers*, have become extremely popular with malware writers because they help malware hide from antivirus software, complicate malware analysis, and shrink the size of a malicious executable. Most packers are easy to use and are freely available. Basic static analysis isn't useful on a packed program; packed malware must be unpacked before it can be analyzed statically, which makes analysis more complicated and challenging.

Packers are used on executables for two main reasons: to shrink programs or to thwart detection or analysis. Even though there are a wide variety of packers, they all follow a similar pattern: They transform an executable to create a new executable that stores the transformed executable as data and contains an unpacking stub that is called by the OS.

We begin this chapter with some background information about how packers work and how to recognize them. Then we will discuss unpacking strategies, starting with simple ones and then moving on to strategies that are progressively more complicated.

Packer Anatomy

When malware has been packed, an analyst typically has access to only the packed file, and cannot examine the original unpacked program or the program that packed the malware. In order to unpack an executable, we must undo the work performed by the packer, which requires that we understand how a packer operates.

All packers take an executable file as input and produce an executable file as output. The packed executable is compressed, encrypted, or otherwise transformed, making it harder to recognize and reverse-engineer.

Most packers use a compression algorithm to compress the original executable. A packer designed to make the file difficult to analyze may encrypt the original executable and employ anti-reverse-engineering techniques, such as anti-disassembly, anti-debugging, or anti-VM. Packers can pack the entire executable, including all data and the resource section, or pack only the code and data sections.

To maintain the functionality of the original program, a packing program needs to store the program's import information. The information can be stored in any format, and there are several common strategies, which are covered in depth later in this chapter. When unpacking a program, reconstructing the import section can sometimes be challenging and time-consuming, but it's necessary for analyzing the program's functionality.

The Unpacking Stub

Unpacked executables are loaded by the OS. With packed programs, the unpacking stub is loaded by the OS, and then the unpacking stub loads the original program. The code entry point for the executable points to the unpacking stub rather than the original code. The original program is generally stored in one or more extra sections of the file.

The unpacking stub can be viewed by a malware analyst, and understanding the different parts of the stub is fundamental to unpacking the executable. The unpacking stub is often small, since it does not contribute to the main functionality of the program, and its function is typically simple: unpack the original executable. If you attempt to perform static analysis on the packed program, you will be analyzing the stub, not the original program.

The unpacking stub performs three steps:

- Unpacks the original executable into memory
- Resolves all of the imports of the original executable
- Transfers execution to the original entry point (OEP)

Loading the Executable

When regular executables load, a loader reads the PE header on the disk, and allocates memory for each of the executable's sections based on that header. The loader then copies the sections into the allocated spaces in memory.