# Malware Data Science

*Attack Detection and Attribution*

Joshua Saxe with Hillary Sanders

*Foreword by Anup Ghosh, PhD*

no starch press

ISBN-10: 1-59327-859-4
ISBN-13: 978-1-59327-859-5

Publisher: William Pollock
Production Editor: Laurel Chun
Cover Illustration: Jonny Thomas
Interior Design: Octopod Studios
Developmental Editors: Annie Choi and William Pollock
Technical Reviewer: Gabor Szappanos
Copyeditor: Barton Reed
Compositor: Laurel Chun
Proofreader: James Fraleigh
Indexer: BIM Creatives, LLC

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

Library of Congress Control Number: 2018949204

# BRIEF CONTENTS

# CONTENTS IN DETAIL

# FOREWORD

Congratulations on picking up *Malware Data Science*. You're on your way to equipping yourself with the skills necessary to become a cybersecurity professional. In this book, you'll find a wonderful introduction to data science as applied to malware analysis, as well as the requisite skills and tools you need to be proficient at it.

There are far more jobs in cybersecurity than there are qualified candidates, so the good news is that cybersecurity is a great field to get into. The bad news is that the skills required to stay current are changing rapidly. As is often the case, necessity is the mother of invention. With far more demand for skilled cybersecurity professionals than there is supply, data science algorithms are filling the gap by providing new insights and predictions about threats against networks. The traditional model of watchmen monitoring network data is rapidly becoming obsolete as data science is increasingly being used to find threat patterns in terabytes of data. And thank goodness for that, because monitoring a screen of alerts is about as exciting as monitoring a video camera surveillance system of a parking lot.

So what exactly is data science and how does it apply to security? As you'll see in the Introduction, data science applied to security is the art and science of using machine learning, data mining, and visualization to detect threats against networks. While you'll find a lot of hyperbole around machine learning and artificial intelligence driven by marketing, there are, in fact, very good use cases for these technologies that are in production today.

For instance, when it comes to malware detection, both the volume of malware production and the cost to the adversary in changing malware signatures has rendered signature-only based approaches to malware obsolete. Instead, antivirus companies are now training neural networks or other types of machine learning algorithms over very large datasets of malware to learn their characteristics, so that new variants of malware can be detected without having to update the model daily. The combination of signature-based and machine learning–based detection provides coverage for both known and unknown malware. This is a topic both Josh and Hillary are experts in and from which they speak from deep experience.

But malware detection is only one use case for data science. In fact, when it comes to finding threats on the network, today's sophisticated adversaries often will not drop executable programs. Instead, they will exploit existing software for initial access and then leverage system tools to pivot from one machine to the next using the user privileges obtained through exploitation. From an adversarial point of view this approach doesn't leave behind artifacts such as malware that antivirus software will detect. However, a good endpoint logging system or an endpoint detection and response (EDR) system will capture system level activities and send this telemetry to the cloud, from where analysts can attempt to piece together the digital footprints of an intruder. This process of combing through massive streams of data and continuously looking for patterns of intrusion is a problem well-suited for data science, specifically data mining with statistical algorithms and data visualization. You can expect more and more Security Operations Centers (SOCs) to adopt data mining and artificial intelligence technologies. It's really the only way to cull through massive data sets of system events to identify actual attacks.

Cybersecurity is undergoing massive shifts in technology and its operations, and data science is driving the change. We are fortunate to have experts like Josh Saxe and Hillary Sanders not only share their expertise with us, but do it in such an engaging and accessible way. This is your opportunity to learn what they know and apply it to your own work so you can stay ahead of the changes in technology and the adversaries you're charged with defeating.

Anup K. Ghosh, PhD
Founder, Invincea, Inc
Washington, DC

# ACKNOWLEDGMENTS

Thanks to Annie Choi, Laurel Chun, and Bill Pollock at No Starch Press and to my copyeditor, Bart Reed. In all justice, they should be regarded as co-authors of this book. Thanks in advance to the workers responsible for printing, transporting, and selling copies of this book, and the engineers responsible for its digital storage, transmission, and rendering. Thanks to Hillary Sanders for bringing her remarkable talents to the project exactly when they were needed. Gratitude to Gabor Szappanos for his excellent and exacting technical review.

Thanks to my two year old daughter Maya, who, I'm happy to share, slowed this project down dramatically. Thanks to Alen Capalik, Danny Hillis, Chris Greamo, Anup Ghosh, and Joe Levy for their mentorship over the past 10 years. Deep appreciation to the Defense Advanced Research Projects Agency (DARPA) and Timothy Fraser for supporting the research on which much of this book is based. Thanks to Mandiant, and Mila Parkour, for obtaining and curating the APT1 malware samples used for demonstration purposes in this book. Deep appreciation to the authors of Python, NetworkX, `matplotlib`, `numpy`, `sklearn`, `Keras`, `seaborn`, `pefile`, `icoutils`, *malwr.com*, CuckooBox, `capstone`, `pandas`, and `sqlite` for your contributions to free and open source security and data science software.

Tremendous gratitude to my parents, Maryl Gearhart and Geoff Saxe, for introducing me to computers, for tolerating my teenage hacker phase (and all the illegality that entailed), and for their boundless love and support. Thanks to Gary Glickman for his indispensable love and support. Finally, thanks to Ksenya Gurshtein, my partner in life, for supporting me in this endeavor completely and without hesitation.

Joshua Saxe

Thanks to Josh, for including me in this! Thanks to Ani Adhikari for being an amazing teacher. Thanks to Jacob Michelini, because he really wanted his name

in a book.

Hillary Sanders

# INTRODUCTION

If you're working in security, chances are you're using data science more than ever before, even if you may not realize it. For example, your antivirus product uses data science algorithms to detect malware. Your firewall vendor may have data science algorithms detecting suspicious network activity. Your security information and event management (SIEM) software probably uses data science to identify suspicious trends in your data. Whether conspicuously or not, the entire security industry is moving toward incorporating more data science into security products.

Advanced IT security professionals are incorporating their own custom machine learning algorithms into their workflows. For example, in recent conference presentations and news articles, security analysts at Target, Mastercard, and Wells Fargo all described developing custom data science technologies that they use as part of their security workflows.[1] If you're not already on the data science bandwagon, there's no better time to upgrade your skills to include data science into your security practice.

## What Is Data Science?

*Data science* is a growing set of algorithmic tools that allow us to understand and make predictions about data using statistics, mathematics, and artful statistical data visualizations. More specific definitions exist, but generally, data science has three subcomponents: machine learning, data mining, and data visualization.

In the security context, machine learning algorithms learn from training data to detect new threats. These methods have been proven to detect malware that flies under the radar of traditional detection techniques like signatures. Data mining algorithms search security data for interesting patterns (such as relationships between threat actors) that might help us discern attack campaigns targeting our organizations. Finally, data visualization renders sterile, tabular data into graphical format to make it easier for people to spot interesting and suspicious trends. I cover all three areas in depth in this book and show you how to apply them.

## Why Data Science Matters for Security

Data science is critically important for the future of cybersecurity for three reasons: first, security is *all about data*. When we seek to detect cyber threats, we're analyzing data in the form of files, logs, network packets, and other artifacts. Traditionally, security professionals didn't use data science techniques to make detections based on these data sources. Instead, they used file hashes, custom-written rules like signatures, and manually defined heuristics. Although these techniques have their merits, they required handcrafted techniques for each type of attack, necessitating too much manual work to keep up with the changing cyber threat landscape. In recent years, data science techniques have become crucial in bolstering our ability to detect threats.

Second, data science is important to cybersecurity because the number of cyberattacks on the internet has grown dramatically. Take the growth of the malware underworld as an example. In 2008, there were about 1 million unique malware executables known to the security community. By 2012, there were 100 million. As this book goes to press in 2018, there are more than 700 million malicious executables known to the security community (*https://www.av-test.org/en/statistics/malware/*), and this number is likely to grow.

Due to the sheer volume of malware, manual detection techniques such as signatures are no longer a reasonable method for detecting all cyberattacks. Because data science techniques automate much of the work that goes into detecting cyberattacks, and vastly decrease the memory usage needed to detect such attacks, they hold tremendous promise in defending networks and users as cyber threats grow.

Finally, data science matters for security because data science is *the* technical trend of the decade, both inside and outside of the security industry, and it will likely remain so through the next decade. Indeed, you've probably seen applications of data science everywhere—in personal voice assistants (Amazon Echo, Siri, and Google Home), self-driving cars, ad recommendation systems, web search engines, medical image analysis systems, and fitness tracking apps.

We can expect data science–driven systems to have major impacts in legal services, education, and other areas. Because data science has become a key enabler across the technical landscape, universities, major companies (Google, Facebook, Microsoft, and IBM), and governments are investing billions of dollars to improve data science tools. Thanks to these investments, data science tools will become even more adept at solving hard attack-detection problems.

## Applying Data Science to Malware

This book focuses on data science as it applies to *malware*, which we define as executable programs written with malicious intent, because malware continues to be the primary means by which threat actors gain a foothold on networks and subsequently achieve their goals. For example, in the ransomware scourge that has emerged in recent years, attackers typically send users malicious email attachments that download ransomware executables (malware) to users' machines, which then encrypt users' data and ask them for a ransom to decrypt the data. Although skilled attackers working for governments sometimes avoid using malware altogether to fly under the radar of detection systems, malware continues to be the major enabling technology in cyberattacks today.

By homing in on a specific application of security data science rather than attempting to cover security data science broadly, this book aims to show more thoroughly how data science techniques can be applied to a major security problem. By understanding malware data science, you'll be better equipped to apply data science to other areas of security, like detecting network attacks, phishing emails, or suspicious user behavior. Indeed, almost all the techniques you'll learn in this book apply to building data science detection and intelligence systems in general, not just for malware.

## Who Should Read This Book?

This book is aimed toward security professionals who are interested in learning more about how to apply data science to computer security problems. If computer security *and* data science are new to you, you might find yourself having to look up terms to give yourself a little bit of context, but you can still read this book successfully. If you're only interested in data science, but not security, this book is probably not for you.

## About This Book

The first part of the book consists of three chapters that cover basic reverse engineering concepts necessary for understanding the malware data science techniques discussed later in the book. If you're new to malware, read the first three chapters first. If you're an old hand at malware reverse engineering, you can skip these chapters.

- **Chapter 1: Basic Static Malware Analysis** covers static analysis techniques for picking apart malware files and discovering how they achieve malicious ends on our computers.
- **Chapter 2: Beyond Basic Static Analysis: x86 Disassembly** gives you a brief overview of x86 assembly language and how to disassemble and reverse engineer malware.
- **Chapter 3: A Brief Introduction to Dynamic Analysis** concludes the reverse engineering section of the book by discussing dynamic analysis, which involves running malware in controlled environments to learn about its behavior.

The next two chapters of the book, Chapters 4 and 5, focus on malware relationship analysis, which involves looking at similarities and differences between collections of malware to identify malware campaigns against your organization, such as a ransomware campaign controlled by a group of cybercriminals, or a concerted, targeted attack on your organization. These stand-alone chapters are for readers who are interested not only in detecting malware, but also in extracting valuable threat intelligence to learn who is attacking their network. If you're less interested in threat intelligence and more interested in data science–driven malware detection, you can safely skip these chapters.

- **Chapter 4: Identifying Attack Campaigns Using Malware Networks** shows you how to analyze and visualize malware based on shared attributes, such as the hostnames that malware programs call out to.
- **Chapter 5: Shared Code Analysis** explains how to identify and visualize shared code relationships between malware samples, which can help you identify whether groups of malware samples came from one or multiple criminal groups.

The next four chapters cover everything you need to know to understand, apply, and implement machine learning–based malware detection systems. These chapters also provide a foundation for applying machine learning to other security contexts.

- **Chapter 6: Understanding Machine Learning–Based Malware Detectors** is an accessible, intuitive, and non-mathematical introduction to basic machine learning concepts. If you have a history with machine learning, this chapter will provide a convenient refresher.
- **Chapter 7: Evaluating Malware Detection Systems** shows you how to evaluate the accuracy of your machine learning systems using basic statistical methods so that you can select the best possible approach.
- **Chapter 8: Building Machine Learning Detectors** introduces open source machine learning tools you can use to build your own machine learning systems and explains how to use them.
- **Chapter 9: Visualizing Malware Trends** covers how to visualize malware threat data to reveal attack campaigns and trends using Python, and how to integrate data visualization into your day-to-day workflow

when analyzing security data.

The last three chapters introduce deep learning, an advanced area of machine learning that involves a bit more math. Deep learning is a hot growth area within security data science, and these chapters provide enough to get you started.

- **Chapter 10: Deep Learning Basics** covers the basic concepts that underlie deep learning.
- **Chapter 11: Building a Neural Network Malware Detector with Keras** explains how to implement deep learning–based malware detection systems in Python using open source tools.
- **Chapter 12: Becoming a Data Scientist** concludes the book by sharing different pathways to becoming a data scientist and qualities that can help you succeed in the field.
- **Appendix: An Overview of Datasets and Tools** describes the data and example tool implementations accompanying the book.

## How to Use the Sample Code and Data

No good programming book is complete without sample code to play with and extend on your own. Sample code and data accompany each chapter of this book and are described exhaustively in the appendix. All the code targets Python 2.7 in Linux environments. To access the code and data, you can download a VirtualBox Linux virtual machine, which has the code, data, and supporting open source tools all set up and ready to go, and run it within your own VirtualBox environment. You can download the book's accompanying data at *http://www.malwaredatascience.com/*, and you can download the VirtualBox for free at *https://www.virtualbox.org/wiki/Downloads*. The code has been tested on Linux, but if you prefer to work outside of the Linux VirtualBox, the same code should work almost as well on MacOS, and to a lesser extent on Windows machines.

If you'd rather install the code and data in your own Linux environment, you can download them here: *http://www.malwaredatascience.com/*. You'll find a directory for each chapter in the downloadable archive, and within each chapter's directory there are *code/* and *data/* directories that contain the corresponding code and data. Code files correspond to chapter listings or sections, whichever makes more sense for the application at hand. Some code files are exactly like the listings, whereas others have been changed slightly to make it easier for you to play with parameters and other options. Code directories come with pip *requirements.txt* files, which give the open source libraries that the code in that chapter depends on to run. To install these libraries on your machine, simply type `pip -r requirements.txt` in each chapter's *code/* directory.

Now that you have access to the code and data for this book, let's get started.

# 1
# BASIC STATIC MALWARE ANALYSIS

In this chapter we look at the basics of static malware analysis. Static analysis is performed by analyzing a program file's disassembled code, graphical images, printable strings, and other on-disk resources. It refers to reverse engineering without actually running the program. Although static analysis techniques have their shortcomings, they can help us understand a wide variety of malware. Through careful reverse engineering, you'll be able to better understand the benefits that malware binaries provide attackers after they've taken possession of a target, as well as the ways attackers can hide and continue their attacks on an infected machine. As you'll see, this chapter combines descriptions and examples. Each section introduces a static analysis technique and then illustrates its application in real-world analysis.

I begin this chapter by describing the Portable Executable (PE) file format used by most Windows programs, and then examine how to use the popular Python library `pefile` to dissect a real-world malware binary. I then describe techniques such as imports analysis, graphical image analysis, and strings analysis. In all cases, I show you how to use open source tools to apply the analysis technique to real-world malware. Finally, at the end of the chapter, I introduce ways malware can make life difficult for malware analysts and discuss some ways to mitigate these issues.

You'll find the malware sample used in the examples in this chapter in this book's data under the directory */ch1*. To demonstrate the techniques discussed in this chapter, we use *ircbot.exe*, an Internet Relay Chat (IRC) bot created for experimental use, as an example of the kinds of malware commonly observed in the wild. As such, the program is designed to stay resident on a target computer while connected to an IRC server. After *ircbot.exe* gets hold of a target, attackers can control the target computer via IRC, allowing them to take actions such as turning on a webcam to capture and surreptitiously extract video feeds of the target's physical location, taking screenshots of the desktop, extracting files from the target machine, and so on. Throughout this chapter, I demonstrate how static analysis techniques can reveal the capabilities of this malware.

## The Microsoft Windows Portable Executable Format

To perform static malware analysis, you need to understand the Windows PE format, which describes the structure of modern Windows program files such as *.exe*, *.dll*, and *.sys* files and defines the way they store data. PE files contain x86 instructions, data such as images and text, and metadata that a program needs in order to run.

The PE format was originally designed to do the following:

**Tell Windows how to load a program into memory** The PE format describes which chunks of a file should be loaded into memory, and where. It also tells you where in the program code Windows should start a program's execution and which dynamically linked code libraries should be loaded into memory.

**Supply media (or resources) a running program may use in the course of its execution** These resources can include strings of characters like the ones in GUI dialogs or console output, as well as images or videos.

**Supply security data such as digital code signatures** Windows uses such security data to ensure that code comes from a trusted source.

The PE format accomplishes all of this by leveraging the series of constructs shown in Figure 1-1.

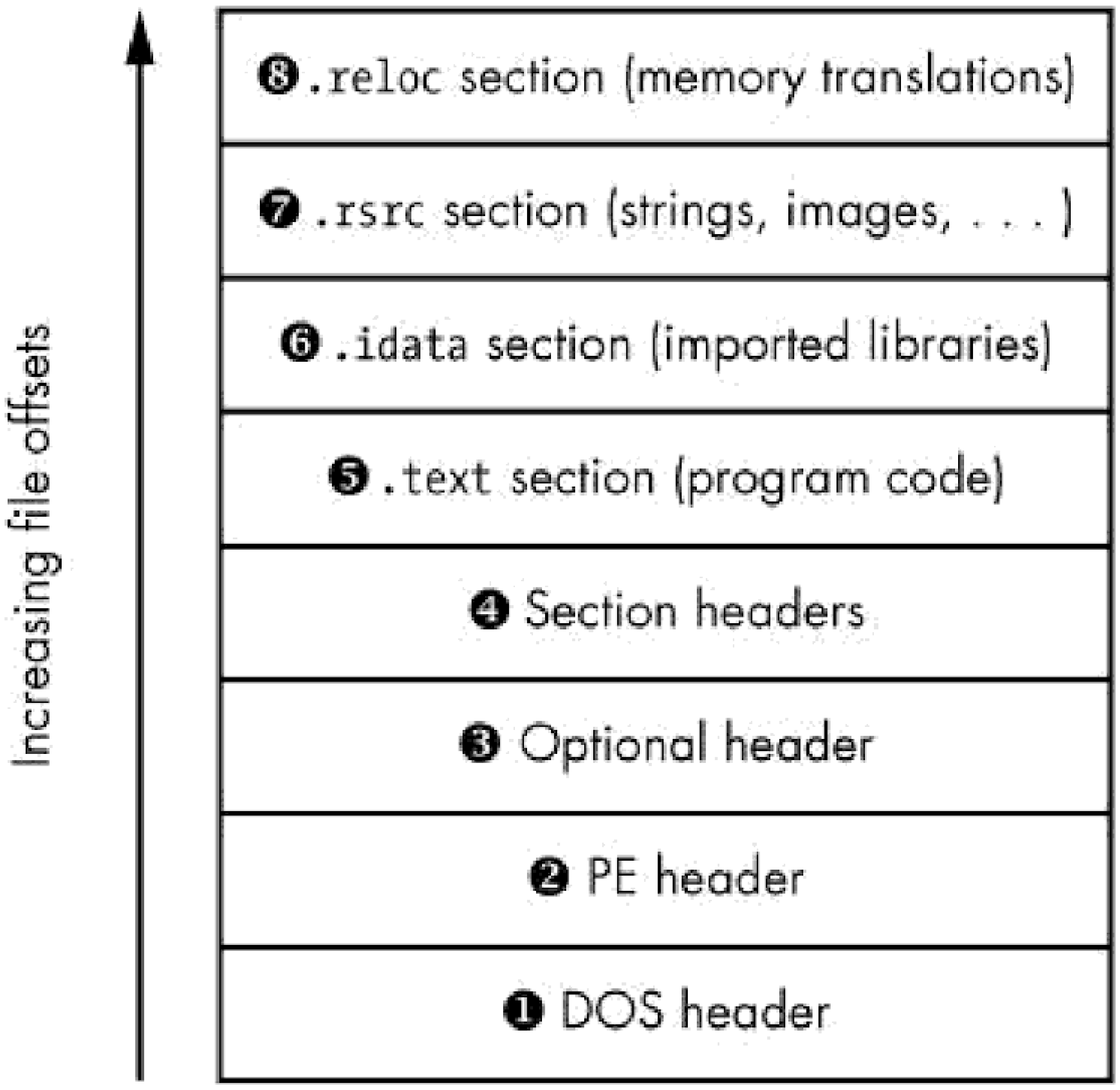

❽ .reloc section (memory translations)
❼ .rsrc section (strings, images, . . . )
❻ .idata section (imported libraries)
❺ .text section (program code)
❹ Section headers
❸ Optional header
❷ PE header
❶ DOS header
Increasing file offsets

*Figure 1-1: The PE file format*

As the figure shows, the PE format includes a series of headers telling the operating system how to load the program into memory. It also includes a series of sections that contain the actual program data. Windows loads the sections into memory such that their memory offsets correspond to where they appear on disk. Let's explore this file structure in more detail, starting with the PE header. We'll skip over a discussion of the DOS header, which is a relic of the 1980s-era Microsoft DOS operating system and only present for compatibility reasons.

### The PE Header

Shown at the bottom of Figure 1-1, above the DOS header ❶, is the PE header ❷, which defines a program's general attributes such as binary code, images, compressed data, and other program attributes. It also tells us whether a program is designed for 32- or 64-bit systems. The PE header provides basic but useful contextual information to the malware analyst. For example, the header includes a timestamp field that can give away the time at which the malware author compiled the file. This happens when malware authors forget to replace this field with a bogus value, which they often do.

## The Optional Header

The optional header ❸ is actually ubiquitous in today's PE executable programs, contrary to what its name suggests. It defines the location of the program's *entry point* in the PE file, which refers to the first instruction the program runs once loaded. It also defines the size of the data that Windows loads into memory as it loads the PE file, the Windows subsystem, the program targets (such as the Windows GUI or the Windows command line), and other high-level details about the program. The information in this header can prove invaluable to reverse engineers, because a program's entry point tells them where to begin reverse engineering.

## Section Headers

Section headers ❹ describe the data sections contained within a PE file. A *section* in a PE file is a chunk of data that either will be mapped into memory when the operating system loads a program or will contain instructions about how the program should be loaded into memory. In other words, a section is a sequence of bytes on disk that will either become a contiguous string of bytes in memory or inform the operating system about some aspect of the loading process.

Section headers also tell Windows what permissions it should grant to sections, such as whether they should be readable, writable, or executable by the program when it's executing. For example, the `.text` section containing x86 code will typically be marked readable and executable but not writable to prevent program code from accidentally modifying itself in the course of execution.

A number of sections, such as `.text` and `.rsrc`, are depicted in Figure 1-1. These get mapped into memory when the PE file is executed. Other special sections, such as the `.reloc` section, aren't mapped into memory. We'll discuss these sections as well. Let's go over the sections shown in Figure 1-1.

### The .text Section

Each PE program contains at least one section of x86 code marked executable in its section header; these sections are almost always named `.text` ❺. We'll disassemble the data in the `.text` section when performing program disassembly and reverse engineering in Chapter 2.

### The .idata Section

The `.idata` section ❻, also called *imports*, contains the *Import Address Table (IAT)*, which lists dynamically linked libraries and their functions. The IAT is among the most important PE structures to inspect when initially approaching a PE binary for analysis because it reveals the library calls a program makes, which in turn can betray the malware's high-level functionality.

### The Data Sections

The data sections in a PE file can include sections like `.rsrc`, `.data`, and `.rdata`, which store items such as mouse cursor images, button skins, audio, and other media used by a program. For example, the `.rsrc` section ❼ in Figure 1-1 contains printable character strings that a program uses to render text as strings.

The information in the `.rsrc` (resources) section can be vital to malware analysts because by examining the printable character strings, graphical images, and other assets in a PE file, they can gain vital clues about the file's functionality. In "Examining Malware Images" on page 7, you'll learn how to use the `icoutils` toolkit (including `icotool` and `wrestool`) to extract graphical images from malware binaries' resources sections. Then, in "Examining Malware Strings" on page 8, you'll learn how to extract printable strings from malware resources sections.

### The .reloc Section

A PE binary's code is not *position independent*, which means it will not execute correctly if it's moved from its intended memory location to a new memory location. The `.reloc` section ❽ gets around this by allowing code to be moved without breaking. It tells the Windows operating system to translate memory addresses in a PE file's code if the code has been moved so that the code still runs correctly. These translations usually involve adding or subtracting an offset from a memory address.

Although a PE file's `.reloc` section may well contain information you'll want to use in your malware analysis, we won't discuss it further in this book because our focus is on applying machine learning and data analysis to malware, not doing the kind of hardcore reverse engineering that involves looking at relocations.

## Dissecting the PE Format Using pefile

The `pefile` Python module, written and maintained by Ero Carerra, has become an industry-standard malware analysis library for dissecting PE files. In this section, I show you how to use `pefile` to dissect *ircbot.exe*. The *ircbot.exe* file can be found on the virtual machine accompanying this book in the directory *~/malware_data_science/ch1/data*. Listing 1-1 assumes that *ircbot.exe* is in your current working directory.

Enter the following to install the `pefile` library so that we can import it within Python:

```
$ pip install pefile
```

Now, use the commands in Listing 1-1 to start Python, import the `pefile` module, and open and parse the PE file *ircbot.exe* using `pefile`.

```
$ python
>>> import pefile
>>> pe = pefile.PE("ircbot.exe")
```

*Listing 1-1: Loading the `pefile` module and parsing a PE file* (ircbot.exe)

We instantiate `pefile.PE`, which is the core class implemented by the PE module. It parses PE files so that we can examine their attributes. By calling the PE constructor, we load and parse the specified PE file, which is *ircbot.exe* in this example. Now that we've loaded and parsed our file, run the code in Listing 1-2 to pull information from *ircbot.exe*'s PE fields.

```
# based on Ero Carrera's example code (pefile library author)
for section in pe.sections:
  print (section.Name, hex(section.VirtualAddress),
    hex(section.Misc_VirtualSize), section.SizeOfRawData )
```

*Listing 1-2: Iterating through the PE file's sections and printing information about them*

Listing 1-3 shows the output.

```
('.text\x00\x00\x00', ❶'0x1000', ❷'0x32830', ❸207360)
('.rdata\x00\x00', '0x34000', '0x427a', 17408)
('.data\x00\x00\x00', '0x39000', '0x5cff8', 10752)
('.idata\x00\x00', '0x96000', '0xbb0', 3072)
('.reloc\x00\x00', '0x97000', '0x211d', 8704)
```

*Listing 1-3: Pulling section data from* ircbot.exe *using Python's* `pefile` *module*

As you can see in Listing 1-3, we've pulled data from five different sections of the PE file: `.text`, `.rdata`, `.data`, `.idata`, and `.reloc`. The output is given as five tuples, one for each PE section pulled. The first entry on each line identifies the PE section. (You can ignore the series of `\x00` null bytes, which are simply C-style null string terminators.) The remaining fields tell us what each section's memory utilization will be once it's loaded into memory and where in memory it will be found once loaded.

For example, `0x1000` ❶ is the *base virtual memory address* where these sections will be loaded. Think of this as the section's base memory address. The `0x32830` ❷ in the *virtual size* field specifies the amount of memory required by the section once loaded. The `207360` ❸ in the third field represents the amount of data the section will take up within that chunk of memory.

In addition to using `pefile` to parse a program's sections, we can also use it to list the DLLs a binary will load, as well as the function calls it will request within those DLLs. We can do this by dumping a PE file's IAT. Listing 1-4 shows how to use `pefile` to dump the IAT for *ircbot.exe*.

```
$ python
pe = pefile.PE("ircbot.exe")
for entry in pe.DIRECTORY_ENTRY_IMPORT:
    print entry.dll
    for function in entry.imports:
        print '\t',function.name
```

*Listing 1-4: Extracting imports from* ircbot.exe

Listing 1-4 should produce the output shown in Listing 1-5 (truncated for brevity).

```
KERNEL32.DLL
      GetLocalTime
      ExitThread
      CloseHandle
    ❶ WriteFile
    ❷ CreateFileA
      ExitProcess
    ❸ CreateProcessA
      GetTickCount
      GetModuleFileNameA
--snip--
```

*Listing 1-5: Contents of the IAT of* ircbot.exe*, showing library functions used by this malware*

As you can see in Listing 1-5, this output is valuable for malware analysis because it lists a rich array of functions that the malware declares and will reference. For example, the first few lines of the output tell us that the malware will write to files using `WriteFile` ❶, open files using the `CreateFileA` call ❷, and create new processes using `CreateProcessA` ❸. Although this is fairly basic information about the malware, it's a start in understanding the malware's behavior in more detail.

## Examining Malware Images

To understand how malware may be designed to game a target, let's look at the icons contained in its `.rsrc` section. For example, malware binaries are often designed to trick users into clicking them by masquerading as Word documents, game installers, PDF files, and so on. You also find images in the malware suggesting programs of interest to the attackers themselves, such as network attack tools and programs run by attackers for the remote control of compromised machines. I have even seen binaries containing desktop icons of jihadists, images of evil-looking cyberpunk cartoon characters, and images of Kalashnikov rifles. For our sample image analysis, let's consider a malware sample the security company Mandiant identified as having been crafted by a Chinese state-sponsored hacking group. You can find this sample malware in this chapter's data directory under the name *fakepdfmalware.exe*. This sample uses an Adobe Acrobat icon to trick users into thinking it is an Adobe Acrobat document, when in fact it's a malicious PE executable.

Before we can extract the images from the *fakepdfmalware.exe* binary using the Linux command line tool `wrestool`, we first need to create a directory to hold the images we'll extract. Listing 1-6 shows how to do all this.

```
$ mkdir images
$ wrestool -x fakepdfmalware.exe -output=images
$ icotool -x -o images images/*.ico
```

*Listing 1-6: Shell commands that extract images from a malware sample*

We first use `mkdir images` to create a directory to hold the extracted images. Next, we use `wrestool` to extract image resources (`-x`) from *fakepdfmalware.exe* to */images* and then use `icotool` to extract (`-x`) and convert (`-o`) any resources in the Adobe *.ico* icon format into *.png* graphics so that we can view them using standard image viewer tools. If you don't have `wrestool` installed on your system, you can download it at *http://www.nongnu.org/icoutils/*.

Once you've used `wrestool` to convert the images in the target executable to the PNG format, you should be able open them in your favorite image viewer and see the Adobe Acrobat icon at various resolutions. As my example here demonstrates, extracting images and icons from PE files is relatively straightforward and can quickly reveal interesting and useful information about malware binaries. Similarly, we can easily extract printable strings from malware for more information, which we'll do next.

## Examining Malware Strings

*Strings* are sequences of printable characters within a program binary. Malware analysts often rely on strings in a malicious sample to get a quick sense of what may be going on inside it. These strings often contain things like HTTP and FTP commands that download web pages and files, IP addresses and hostnames that tell you what addresses the malware connects to, and the like. Sometimes even the language used to write the strings can hint at a malware binary's country of origin, though this can be faked. You may even find text in a string that explains in leetspeak the purpose of a malicious binary.

Strings can also reveal more technical information about a binary. For example, you may find information about the compiler used to create it, the programming language the binary was written in, embedded scripts or HTML, and so on. Although malware authors can obfuscate, encrypt, and compress all of these traces, even advanced malware authors often leave at least some traces exposed, making it particularly important to examine `strings` dumps when analyzing malware.

### *Using the strings Program*

The standard way to view all strings in a file is to use the command line tool `strings`, which uses the following syntax:

```
$ strings filepath | less
```

This command prints all strings in a file to the terminal, line by line. Adding `| less` at the end prevents the strings from just scrolling across the terminal. By default, the `strings` command finds all printable strings with a minimum length of 4 bytes, but you can set a different minimum length and change various other parameters, as listed in the commands manual page. I recommend simply using the default minimum string length of 4, but you can change the minimum string length using the `-n` option. For example, `strings -n 10 filepath` would extract only strings with a minimum length of 10 bytes.

## *Analyzing Your strings Dump*

Now that we dumped a malware program's printable strings, the challenge is to understand what the strings mean. For example, let's say we dump the strings to the *ircbotstring.txt* file for *ircbot.exe*, which we explored earlier in this chapter using the `pefile` library, like this:

```
$ strings ircbot.exe > ircbotstring.txt
```

The contents of *ircbotstring.txt* contain thousands of lines of text, but some of these lines should stick out. For example, Listing 1-7 shows a bunch of lines extracted from the string dump that begin with the word `DOWNLOAD`.

```
[DOWNLOAD]: Bad URL, or DNS Error: %s.
[DOWNLOAD]: Update failed: Error executing file: %s.
[DOWNLOAD]: Downloaded %.1fKB to %s @ %.1fKB/sec. Updating.
[DOWNLOAD]: Opened: %s.
--snip--
[DOWNLOAD]: Downloaded %.1f KB to %s @ %.1f KB/sec.
[DOWNLOAD]: CRC Failed (%d != %d).
[DOWNLOAD]: Filesize is incorrect: (%d != %d).
[DOWNLOAD]: Update: %s (%dKB transferred).
[DOWNLOAD]: File download: %s (%dKB transferred).
[DOWNLOAD]: Couldn't open file: %s.
```

*Listing 1-7: The `strings` output showing evidence that the malware can download files specified by the attacker onto a target machine*

These lines indicate that *ircbot.exe* will attempt to download files specified by an attacker onto the target machine.

Let's try analyzing another one. The string dump shown in Listing 1-8 indicates that *ircbot.exe* can act as a web server that listens on the target machine for connections from the attacker.

```
❶ GET
❷ HTTP/1.0 200 OK
```

```
  Server: myBot
  Cache-Control: no-cache,no-store,max-age=0
  pragma: no-cache
  Content-Type: %s
  Content-Length: %i
  Accept-Ranges: bytes
  Date: %s %s GMT
  Last-Modified: %s %s GMT
  Expires: %s %s GMT
  Connection: close
  HTTP/1.0 200 OK
❸ Server: myBot
  Cache-Control: no-cache,no-store,max-age=0
  pragma: no-cache
  Content-Type: %s
  Accept-Ranges: bytes
  Date: %s %s GMT
  Last-Modified: %s %s GMT
  Expires: %s %s GMT
  Connection: close
  HH:mm:ss
  ddd, dd MMM yyyy
  application/octet-stream
  text/html
```

*Listing 1-8: The `strings` output showing that the malware has an HTTP server to which the attacker can connect*

Listing 1-8 shows a wide variety of HTTP boilerplates used by *ircbot.exe* to implement an HTTP server. It's likely that this HTTP server allows the attacker to connect to a target machine via HTTP to issue commands, such as the command to take a screenshot of the victim's desktop and send it back to the attacker. We see evidence of HTTP functionality throughout the listing. For example, the `GET` method ❶ requests data from an internet resource. The line `HTTP/1.0 200 OK` ❷ is an HTTP string that returns the status code `200`, indicating that all went well with an HTTP network transaction, and `Server: myBot` ❸ indicates that the name of the HTTP server is *myBot*, a giveaway that *ircbot.exe* has a built-in HTTP server.

All of this information is useful in understanding and stopping a particular malware sample or malicious campaign. For example, knowing that a malware sample has an HTTP server that outputs certain strings when you connect to it allows you to scan your network to identify infected hosts.

## Summary

In this chapter, you got a high-level overview of static malware analysis, which involves inspecting a malware program without actually running it. You learned about the PE file format that defines Windows *.exe* and *.dll* files, and you learned how to use the Python library `pefile` to dissect a real-world malware *ircbot.exe* binary. You also used static analysis techniques such as image analysis and strings analysis to extract more information from malware samples. Chapter 2 continues our discussion of static malware analysis with a focus on analyzing the assembly code that can be recovered from malware.

# 2
# BEYOND BASIC STATIC ANALYSIS: X86 DISASSEMBLY

To thoroughly understand a malicious program, we often need to go beyond basic static analysis of its sections, strings, imports, and images. This involves reverse engineering a program's assembly code. Indeed, disassembly and reverse engineering lie at the heart of deep static analysis of malware samples.

Because reverse engineering is an art, technical craft, and science, a thorough exploration is beyond the scope of this chapter. My goal here is to introduce you to reverse engineering so that you can apply it to malware data science. Understanding this methodology is essential for successfully applying machine learning and data analysis to malware.

In this chapter I start with the concepts you'll need to understand x86 disassembly. Later in the chapter I show how malware authors attempt to bypass disassembly and discuss ways to mitigate these anti-analysis and anti-detection maneuvers. But first, let's review some common disassembly methods as well as the basics of x86 assembly language.

## Disassembly Methods

*Disassembly* is the process by which we translate malware's binary code into valid x86 assembly language. Malware authors generally write malware programs in a high-level language like C or C++ and then use a compiler to compile the source code into x86 binary code. Assembly language is the human-readable representation of this binary code. Therefore, disassembling a malware program into assembly language is necessary to understand how it behaves at its core.

Unfortunately, disassembly is no easy feat because malware authors regularly employ tricks to thwart would-be reverse engineers. In fact, perfect disassembly in the face of deliberate obfuscation is an unsolved problem in computer science. Currently, only approximate, error-prone methods exist for disassembling such programs.

For example, consider the case of *self-modifying code*, or binary code that modifies itself as it executes. The only way to disassemble this code properly is to understand the program logic by which the code modifies itself, but that can be exceedingly complex.

Because perfect disassembly is currently impossible, we must use imperfect methods to accomplish this task. The method we'll use is *linear disassembly*, which involves identifying the contiguous sequence of bytes in the Portable Executable (PE) file that corresponds to its x86 program code and then decoding these bytes. The key limitation of this approach is that it ignores subtleties about how instructions are decoded by the CPU in the course of program execution. Also, it doesn't account for the various obfuscations malware authors sometimes use to make their programs harder to analyze.

The other methods of reverse engineering, which we won't cover here, are the more complex disassembly methods used by industrial-grade disassemblers such as IDA Pro. These more advanced methods actually simulate or reason about program execution to discover which assembly instructions a program might reach as a result of a series of conditional branches.

Although this type of disassembly can be more accurate than linear disassembly, it's far more CPU intensive than linear disassembly methods, making it less suitable for data science purposes where the focus is on disassembling thousands or even millions of programs.

Before you can begin analysis using linear disassembly, however, you'll need to review the basic components of assembly language.

## Basics of x86 Assembly Language

Assembly language is the lowest-level human-readable programming language for a given architecture, and it maps closely to the binary instruction format of a particular CPU architecture. A line of assembly language is almost always equivalent to a single CPU instruction. Because assembly is so low level, you can often retrieve it easily from a malware binary by using the right tools.

Gaining basic proficiency in reading disassembled malware x86 code is easier than you might think. This is because most malware assembly code spends most of its time calling into the operating system by way of the Windows operating system's *dynamic-link libraries (DLLs)*, which are loaded into program memory at runtime. Malware programs use DLLs to do most of the real work, such as modifying the system registry, moving and copying files, making network connections and communicating via network protocols, and so on. Therefore, following malware assembly code often involves understanding the ways in which function calls are made from assembly and understanding what various DLL calls do. Of course, things can get much more complicated, but knowing this much can reveal a lot about the malware.

In the following sections I introduce some important assembly language concepts. I also explain some abstract concepts like control flow and control flow graphs. Finally, we disassemble the *ircbot.exe* program and explore how its assembly and control flow can give us insight into its purpose.

There are two major dialects of x86 assembly: Intel and AT&T. In this book I use Intel syntax, which can be obtained from all major disassemblers and is the syntax used in the official Intel documentation of the x86 CPU.

Let's start by taking a look at CPU registers.

### *CPU Registers*

*Registers* are small data storage units on which x86 CPUs perform computations. Because registers are located on the CPU itself, register access is orders of magnitude faster than memory access. This is why core computational operations, such as arithmetic and condition testing instructions, all target registers. It's also why the CPU uses registers to store information about the status of running programs. Although many registers are available to experienced x86 assembly programmers, we'll just focus on a few important ones here.

## General-Purpose Registers

General-purpose registers are like scratch space for assembly programmers. On a 32-bit system, each of these registers contains 32, 16, or 8 bits of space against which we can perform arithmetic operations, bitwise operations, byte order–swapping operations, and more.

In common computational workflows, programs move data into registers from memory or from external hardware devices, perform some operations on this data, and then move the data back out to memory for storage. For example, to sort a long list, a program typically pulls list items in from an array in memory, compares them in the registers, and then writes the comparison results back out to memory.

To understand some of the nuances of the general-purpose register model in the Intel 32-bit architecture, take a look at Figure 2-1.

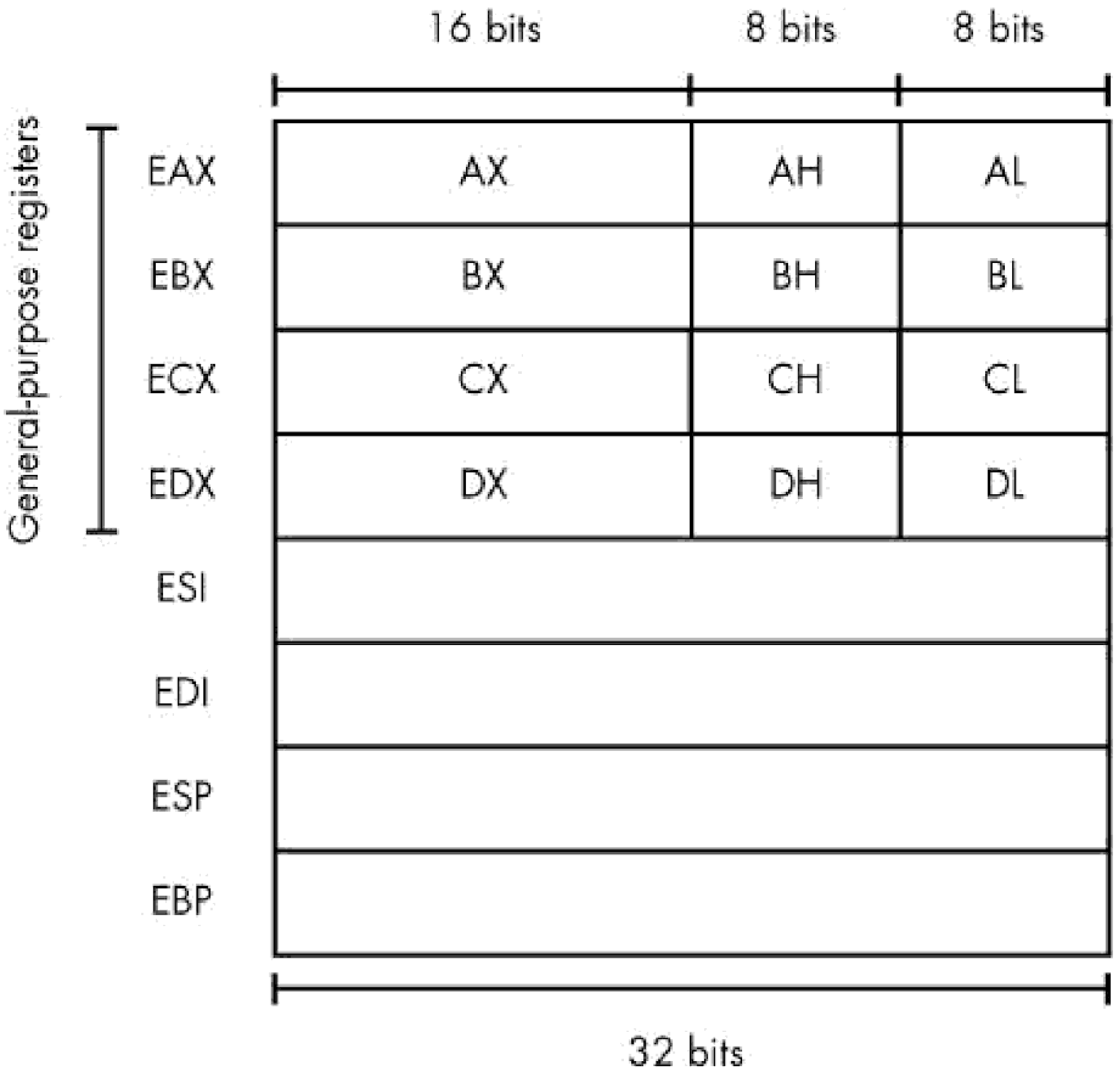


*Figure 2-1: Registers in the x86 architecture*

The vertical axis shows the layout of the general-purpose registers, and the horizontal axis shows how EAX, EBX, ECX, and EDX are subdivided. EAX, EBX, ECX, and EDX are 32-bit registers that have smaller, 16-bit registers inside them: AX, BX, CX, and DX. As you can see in the figure, these 16-bit registers can be subdivided into upper and lower 8-bit registers: AH, AL, BH, BL, CH, CL, DH, and DL. Although it's sometimes useful to address the subdivisions in EAX, EBX, ECX, and EDX, you'll mostly see direct references to EAX, EBX, ECX, and EDX.

## Stack and Control Flow Registers

The stack management registers store critical information about the *program stack*, which is responsible for storing local variables for functions, arguments passed into functions, and control information relating to the program control flow. Let's go over some of these registers.

In simple terms, the ESP register points to the top of the stack for the currently executing function, whereas the EBP register points to the bottom of the stack for the currently executing function. This is crucial information for modern programs, because it means that by referencing data relative to the stack rather than using its absolute address, procedural and object-oriented code can access local variables more gracefully and efficiently.

Although you won't see direct references to the EIP register in x86 assembly code, it's important in security analysis, particularly in the context of vulnerability research and buffer-overflow exploit development. This is because EIP contains the memory address of the currently executing instruction. Attackers can use buffer-overflow exploits to corrupt the value of the EIP register indirectly and take control of program execution.

In addition to its role in exploitation, EIP is also important in the analysis of malicious code deployed by malware. Using a debugger we can inspect EIP's value at any moment, which helps us understand what code malware is executing at any particular time.

EFLAGS is a status register that contains CPU *flags*, which are bits that store status information about the state of the currently executing program. The EFLAGS register is central to the process of making *conditional branches*, or changes in execution flow resulting from the outcome of if/then-style program logic, within x86 programs. Specifically, whenever an x86 assembly program checks whether some value is greater or less than zero and then jumps to a function based on the outcome of this test, the EFLAGS register plays an enabling role, as described in more detail in "Basic Blocks and Control Flow Graphs" on page 19.

## Arithmetic Instructions

*Instructions* operate on general-purpose registers. You can perform simple computations with the general-purpose registers using arithmetic instructions. For example, `add`, `sub`, `inc`, `dec`, and `mul` are examples of arithmetic instructions you'll encounter frequently in malware reverse engineering. Table 2-1 lists some examples of basic instructions and their syntax.

**Table 2-1:** Arithmetic Instructions

| Instructions | Description |
|---|---|
| `add ebx, 100` | Adds 100 to the value in EBX and then stores the result in EBX |
| `sub ebx, 100` | Subtracts 100 from the value in EBX and then stores the result in EBX |
| `inc ah` | Increments the value in AH by 1 |
| `dec al` | Decrements the value in AL by 1 |

The `add` instruction adds two integers and stores the result in the first operand specified, whether this is a memory location or a register according to the following syntax. Keep in mind only one argument can be a memory location. The `sub` instruction is similar to `add`, except it subtracts integers. The `inc` instruction increments a register or memory location's integer value, whereas `dec` decrements a register or memory location's integer value.

## *Data Movement Instructions*

The x86 processor provides a robust set of instructions for moving data between registers and memory. These instructions provide the underlying mechanisms that allow us to manipulate data. The staple memory movement instruction is the `mov` instruction. Table 2-2 shows how you can use the `mov` instruction to move data around.

**Table 2-2:** Data Movement Instructions

| Instructions | Description |
|---|---|
| `mov ebx,eax` | Moves the value in register EAX into register EBX |
| `mov eax, [0x12345678]` | Moves the data at memory address 0x12345678 into the EAX register |
| `mov edx, 1` | Moves the value 1 into the register EDX |
| `mov [0x12345678], eax` | Moves the value in EAX into the memory location 0x12345678 |

Related to the `mov` instruction, the `lea` instruction loads the absolute memory address specified into the register used for getting a pointer to a memory location. For example, `lea edx, [esp-4]` subtracts 4 from the value in ESP and loads the resulting value into EDX.

## Stack Instructions

The *stack* in x86 assembly is a data structure that allows you to push and pop values onto and off of it. This is similar to how you would add and remove plates on and off the top of a stack of plates.

Because control flow is often expressed through C-style function calls in x86 assembly and because these function calls use the stack to pass arguments, allocate local variables, and remember what part of the program to return to after a function finishes executing, the stack and control flow need to be understood together.

The `push` instruction pushes values onto the program stack when the programmer wants to save a register value onto the stack, and the `pop` instruction deletes values from the stack and places them into a designated register.

The `push` instruction uses the following syntax to perform its operations:

```
push 1
```

In this example, the program points the stack pointer (the register ESP) to a new memory address, thereby making room for the value (1), which is now stored at the top location on the stack. Then it copies the value from the argument to the memory location the CPU has just made room for on the top of the stack.

Let's contrast this with `pop`:

```
pop eax
```

The program uses `pop` to pop the top value off the stack and move it into a specified register. In this example, `pop eax` pops the top value off the stack and moves it into `eax`.

An unintuitive but important detail to understand about the x86 program stack is that it grows downward in memory, so that the highest value on the stack is actually stored at the lowest address in stack memory. This becomes very important to remember when you analyze assembly code that references data stored on the stack, as it can quickly get confusing unless you know the stack's memory layout.

Because the x86 stack grows downward in memory, when the `push` instruction allocates space on the program stack for a new value, it decrements the value of ESP so that it points to a lower location in memory and then copies a value from the target register into that memory location, starting at the top address of the stack and growing up. Conversely, the `pop` instruction actually copies the top value off of the stack and then increments the value of ESP so it points to a higher memory location.

## Control Flow Instructions

An x86 program's *control flow* defines the network of possible instruction execution sequences a program may execute, depending on the data, device interactions, and other inputs the program might receive. Control flow instructions define a program's control flow. They are more complicated than stack instructions but still quite intuitive. Because control flow is often expressed through C-style function calls in x86 assembly, the stack and control flow are closely related. They're also related because these function calls use the stack to pass arguments, allocate local variables, and remember what part of the program to return to after a function finishes executing.

The `call` and `ret` control flow instructions are the most important in terms of how programs call functions in x86 assembly and how programs return from functions after these functions are done executing.

The `call` instruction calls a function. Think of this as a function you might write in a higher-level language like C to allow the program to return to the instruction after the `call` instruction is invoked and the function has finished executing. You can invoke the `call` instruction using the following syntax, where *`address`* denotes the memory location where the function's code begins:

```
call address
```

The `call` instruction does two things. First, it pushes the address of the instruction that will execute after the function call returns onto the top of the stack so that the program knows what address to return to after the called function finishes executing. Second, `call` replaces the current value of EIP with the value specified by the *`address`* operand. Then, the CPU begins execution at the new memory location pointed to by EIP.

Just as `call` initiates a function call, the `ret` instruction completes it. You can use the `ret` instruction on its own and without any parameter, as shown here:

```
ret
```

When invoked, `ret` pops the top value off the stack, which we expect to be the saved program counter value (EIP) that the `call` instruction pushed onto the stack when the `call` instruction was invoked. Then it places the popped program counter value back into EIP and resumes execution.

The `jmp` instruction is another important control flow construction, which operates more simply than `call`. Instead of worrying about saving EIP, `jmp` simply tells the CPU to move to the memory address specified as its parameter and begin execution there. For example, `jmp 0x12345678` tells the CPU to start executing the program code stored at memory location 0x12345678 on the next instruction.

You may be wondering how you can make `jmp` and `call` instructions execute in a conditional way, such as "if the program has received a network packet, execute the following function." The answer is that x86 assembly doesn't have high-level constructs like if, then, else, else if, and so on. Instead, branching to an address within a program's code typically requires two instructions: a `cmp` instruction, which checks the value in some register against some test value and stores the result of that test in the EFLAGS register, and a conditional branch instruction.

Most conditional branch instructions start with a *j*, which allows the program to jump to a memory address, and are post-fixed with letters that stand for the condition being tested. For example, `jge` tells the program to jump if greater than or equal to. This means that the value in the register being tested must be greater than or equal to the test value.

The `cmp` instruction uses the following syntax:

```
cmp register, memory location, or literal, register, memory location, or
```

```
literal
```

As stated earlier, `cmp` compares the value in the specified general-purpose register with *`value`* and then stores the result of that comparison in the EFLAGS register.

The various conditional `jmp` instructions are then invoked as follows:

```
j* address
```

As you can see, we can prefix *j* to any number of conditional test instructions. For example, to jump only if the value tested is greater than or equal to the value in the register, use the following instruction:

```
jge address
```

Note that unlike the case of the `call` and `ret` instructions, the `jmp` family of instructions never touches the program stack. In fact, in the case of the `jmp` family of instructions, the x86 program is responsible for tracking its own execution flow and potentially saving or deleting information about what addresses it has visited and where it should return to after a particular sequence of instructions has executed.

## Basic Blocks and Control Flow Graphs

Although x86 programs look sequential when we scroll through their code in a text editor, they actually have loops, conditional branches, and unconditional branches (control flow). All of these give each x86 program a *network* structure. Let's use the simple toy assembly program in Listing 2-1 to see how this works.

```
   setup: # symbol standing in for address of instruction on the next line
❶ mov eax, 10
   loopstart: # symbol standing in for address of the instruction on the next
   line
❷ sub eax, 1
❸ cmp 0, eax
   jne $loopstart
   loopend: # symbol standing in for address of the instruction on the next
   line
```

```
mov eax, 1
# more code would go here
```

*Listing 2-1: Assembly program for understanding control flow graph*

As you can see, this program initializes a counter to the value 10, stored in register EAX ❶. Next, it does a loop in which the value in EAX is decremented by 1 ❷ on each iteration. Finally, once EAX has reached a value of 0 ❸, the program breaks out of the loop.

In the language of control flow graph analysis, we can think of these instructions as comprising three basic blocks. A *basic block* is a sequence of instructions that we know will always execute contiguously. In other words, a basic block always ends with either a branching instruction or an instruction that is the target of a branch, and it always begins with either the first instruction of the program, called the program's *entry point*, or a branch target.

In Listing 2-1, you can see where the basic blocks of our simple program begin and end. The first basic block is composed of the instruction `mov eax, 10` under `setup:`. The second basic block is composed of lines beginning with `sub eax, 1` through `jne $loopstart` under `loopstart:`, and the third starts at `mov eax, 1` under `loopend:`. We can visualize the relationships between the basic blocks using the graph in Figure 2-2. (We use the term *graph* synonymously with the term *network*; in computer science, these terms are interchangeable.)

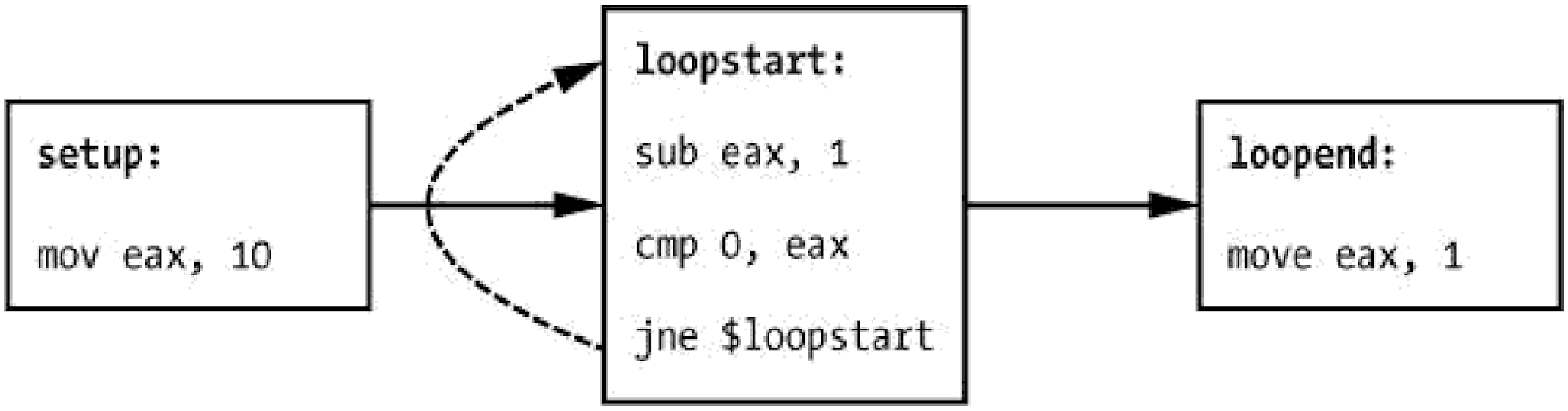


*Figure 2-2: A visualization of the control flow graph of our simple assembly program*

If one basic block can ever flow into another basic block, we connect it, as shown in Figure 2-2. The figure shows that the `setup` basic block leads to the `loopstart` basic block, which repeats 10 times before it transitions to the `loopend` basic block. Real-world programs have control flow graphs such as these, but they're much more complicated, with thousands of basic blocks and thousands of interconnections.

## Disassembling ircbot.exe Using pefile and capstone

Now that you have a good understanding of the basics of assembly language, let's disassemble the first 100 bytes of *ircbot.exe*'s assembly code using linear disassembly. To do this, we'll use the open source Python libraries `pefile` (introduced in Chapter 1) and `capstone`, which is an open source disassembly library that can disassemble 32-bit x86 binary code. You can install both of these libraries with `pip` using the following commands:

```
pip install pefile
pip install capstone
```

Once these two libraries are installed, we can leverage them to disassemble *ircbot.exe* using the code in Listing 2-2.

```
#!/usr/bin/python
import pefile
from capstone import *

# load the target PE file
pe = pefile.PE("ircbot.exe")

# get the address of the program entry point from the program header
entrypoint = pe.OPTIONAL_HEADER.AddressOfEntryPoint

# compute memory address where the entry code will be loaded into memory
entrypoint_address = entrypoint+pe.OPTIONAL_HEADER.ImageBase

# get the binary code from the PE file object
binary_code = pe.get_memory_mapped_image()[entrypoint:entrypoint+100]

# initialize disassembler to disassemble 32 bit x86 binary code
disassembler = Cs(CS_ARCH_X86, CS_MODE_32)
```

```
# disassemble the code
for instruction in disassembler.disasm(binary_code, entrypoint_address):
    print "%s\t%s" %(instruction.mnemonic, instruction.op_str)
```

*Listing 2-2: Disassembling* ircbot.exe

This should produce the following output:

```
❶ push    ebp
  mov     ebp, esp
  push    -1
  push    0x437588
  push    0x41982c
❷ mov     eax, dword ptr fs:[0]
  push    eax
  mov     dword ptr fs:[0], esp
❸ add     esp, -0x5c
  push    ebx
  push    esi
  push    edi
  mov     dword ptr [ebp - 0x18], esp
❹ call    dword ptr [0x496308]
  --snip--
```

Don't worry about understanding all of the instructions in the disassembly output: that would involve an understanding of assembly that goes beyond the scope of this book. However, you should feel comfortable with many of the instructions in the output and have some sense of what they do. For example, the malware pushes the value in register EBP onto the stack ❶, saving its value. Then it proceeds to move the value in ESP into EBP and pushes some numerical values onto the stack. The program moves some data in memory into the EAX register ❷, and it adds the value -0x5c to the value in the ESP register ❸. Finally, the program uses the `call` instruction to call a function stored at the memory address 0x496308 ❹.

Because this is not a book on reverse engineering, I won't go into any more depth here about what the code means. What I've presented is a start to understanding how assembly language works. For more information on assembly language, I recommend the Intel programmer's manual at *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*.

## Factors That Limit Static Analysis

In this chapter and Chapter 1, you learned about a variety of ways in which static analysis techniques can be used to elucidate the purpose and methods of a newly discovered malicious binary. Unfortunately, static analysis has limitations that render it less useful in some circumstances. For example, malware authors can employ certain offensive tactics that are far easier to implement than to defend against. Let's take a look at some of these offensive tactics and see how to defend against them.

### Packing

Malware *packing* is the process by which malware authors compress, encrypt, or otherwise mangle the bulk of their malicious program so that it appears inscrutable to malware analysts. When the malware is run, it unpacks itself and then begins execution. The obvious way around malware packing is to actually run the malware in a safe environment, a dynamic analysis technique I'll cover in Chapter 3.

**NOTE**

*Software packing is also used by benign software installers for legitimate reasons. Benign software authors use packing to deliver their code because it allows them to compress program resources to reduce software installer download sizes. It also helps them thwart reverse engineering attempts by business competitors, and it provides a convenient way to bundle many program resources within a single installer file.*

### Resource Obfuscation

Another anti-detection, anti-analysis technique malware authors use is *resource obfuscation*. They obfuscate the way program resources, such as strings and graphical images, are stored on disk, and then deobfuscate them at runtime so they can be used by the malicious program. For example, a simple obfuscation would be to add a value of 1 to all bytes in images and strings stored in the PE resources section and then subtract 1 from all of this data at runtime. Of course, any number of obfuscations are possible here, all of which make life difficult for malware analysts attempting to make sense of a malware binary using static analysis.

As with packing, one way around resource obfuscation is to just run the malware in a safe environment. When this is not an option, the only mitigation for resource obfuscation is to actually figure out the ways in which malware has obfuscated its resources and to manually deobfuscate them, which is what professional malware analysts often do.

### Anti-disassembly Techniques

A third group of anti-detection, anti-analysis techniques used by malware authors are *anti-disassembly* techniques. These techniques are designed to exploit the inherent limitations of state-of-the-art disassembly techniques to hide code from malware analysts or make malware analysts think that a block of code stored on disk contains different instructions than it actually does.

An example of an anti-disassembly technique involves branching to a memory location that the malware author's disassemblers will interpret as a different instruction, essentially hiding the malware's true instructions from reverse engineers. Anti-disassembly techniques have huge potential and there's no perfect way to defend against them. In practice, the two main defenses against these techniques are to run malware samples in a dynamic environment and to manually figure out where anti-disassembly strategies manifest within a malware sample and how to bypass them.

### Dynamically Downloaded Data

A final class of anti-analysis techniques malware authors use involves externally sourcing data and code. For example, a malware sample may load code dynamically from an external server at malware startup time. If this is the case, static analysis will be useless against such code. Similarly, malware may source decryption keys from external servers at startup time and then use these keys to decrypt data or code that will be used in the malware's execution.

Obviously, if the malware is using an industrial-strength encryption algorithm, static analysis will not be sufficient to recover the encrypted data and code. Such anti-analysis and anti-detection techniques are quite powerful, and the only way around them is to acquire the code, data, or private keys on the external servers by some means and then use them in one's analysis of the malware in question.

## Summary

This chapter introduced x86 assembly code analysis and demonstrated how we can perform disassembly-based static analysis on *ircbot.exe* using open source Python tools. Although this is not meant to be a complete primer on x86 assembly, you should now feel comfortable enough that you have a starting place for figuring out what's going on in a given malware assembly dump. Finally, you learned ways in which malware authors can defend against disassembly and other static analysis techniques, and how you can mitigate these anti-analysis and anti-detection strategies. In Chapter 3, you'll learn to conduct dynamic malware analysis that makes up for many of the weaknesses of static malware analysis.

# 3
# A BRIEF INTRODUCTION TO DYNAMIC ANALYSIS

In Chapter 2, you learned advanced static analysis techniques to disassemble the assembly code recovered from malware. Although static analysis can be an efficient way to gain useful information about malware by studying its different components on disk, it doesn't allow us to observe malware behavior.

In this chapter, you'll learn about the basics of dynamic malware analysis. Unlike static analysis, which focuses on what malware looks like in file form, dynamic analysis consists of running malware in a safe, contained environment to see how it behaves. This is like introducing a dangerous bacterial strain into a sealed environment to see its effects on other cells.

Using dynamic analysis, we can get around common static analysis hurdles, such as packing and obfuscation, as well as gain more direct insight into the purpose of a given malware sample. We begin by exploring basic dynamic analysis techniques, their relevance to malware data science, and their applications. We use open source tools like *malwr.com* to study examples of dynamic analysis in action. Note that this is a condensed survey of the topic and is not intended to be comprehensive. For a more complete introduction, check out *Practical Malware Analysis* (No Starch Press, 2012).

## Why Use Dynamic Analysis?

To understand why dynamic analysis matters, let's consider the problem of packed malware. Recall that packing malware refers to compressing or obfuscating a malware's x86 assembly code to hide the malicious nature of the program. A packed malware sample unpacks itself when it infects a target machine so that the code can execute.

We could try to disassemble a packed or obfuscated malware sample using the static analysis tools discussed in Chapter 2, but this is a laborious process. For example, with static analysis we'd first have to find the location of the obfuscated code in the malware file. Then we'd have to find the location of the deobfuscation subroutines that deobfuscate this code so that it can run. After locating the subroutines, we'd have to figure out how this deobfuscation procedure works in order to perform it on the code. Only then could we begin the actual process of reverse engineering the malicious code.

A simple yet clever alternative to this process is to execute the malware in a safe, contained environment called a *sandbox*. Running malware in a sandbox allows it to unpack itself as it would when infecting a real target. By simply running malware, we can find out what servers a particular malware binary connects to, what system configuration parameters it changes, and what device I/O (input/output) it attempts to perform.

## Dynamic Analysis for Malware Data Science

Dynamic analysis is useful not only for malware reverse engineering but also for malware data science. Because dynamic analysis reveals what a malware sample *does*, we can compare its actions to those of other malware samples. For example, because dynamic analysis shows what files malware samples write to disk, we can use this data to connect those malware samples that write similar filenames to disk. These kinds of clues help us categorize malware samples based on common traits. They can even help us identify malware samples that were authored by the same groups or are part of the same campaigns.

Most importantly, dynamic analysis is useful for building machine learning–based malware detectors. We can train a detector to distinguish between malicious and benign binaries by observing their behaviors during dynamic analysis. For example, after observing thousands of dynamic analysis logs from both malware and benign files, a machine learning system can learn that when *msword.exe* launches a process named *powershell.exe*, this action is malicious, but that when *msword.exe* launches Internet Explorer, this is probably harmless. Chapter 8 will go into more detail about how we can build malware detectors using data based on both static and dynamic analysis. But before we create sophisticated malware detectors, let's look at some basic tools for dynamic analysis.

## Basic Tools for Dynamic Analysis

You can find a number of free, open source tools for dynamic analysis online. This section focuses on *malwr.com* and CuckooBox. The *malwr.com* site has a web interface that allows you to submit binaries for dynamic analysis for free. CuckooBox is a software platform that lets you set up your own dynamic analysis environment so that you can analyze binaries locally. The creators of the CuckooBox platform also operate *malwr.com*, and *malwr.com* runs CuckooBox behind the scenes. Therefore, learning how to analyze results on *malwr.com* will allow you to understand CuckooBox results.

**NOTE**

*At print time,* malwr.com*'s CuckooBox interface was down for maintenance. Hopefully by the time you read this section the site will be back up. If not, the information provided in this chapter can be applied to output from your own CuckooBox instance, which you can set up by following the instructions at* https://cuckoosandbox.org/.

### *Typical Malware Behaviors*

The following are the major categories of actions a malware sample may take upon execution:

**Modifying the file system** For example, writing a device driver to disk, changing system configuration files, adding new programs to the file system, and modifying registry keys to ensure the program auto-starts

**Modifying the Windows registry to change the system configuration** For example, changing firewall settings

**Loading device drivers** For example, loading a device driver that records user keystrokes

**Network actions** For example, resolving domain names and making HTTP requests

We'll examine these behaviors in more detail using a malware sample and analyzing its report on *malwr.com*.

## *Loading a File on malwr.com*

To run a malware sample through *malwr.com*, navigate to *https://malwr.com/* and then click the **Submit** button to upload and submit a binary for analysis. We'll use a binary whose SHA256 hash starts with the characters *d676d95*, which you can find in the data directory accompanying this chapter. I encourage you to submit this binary to *malwr.com* and inspect the results yourself as we go. The submit page is shown in Figure 3-1.

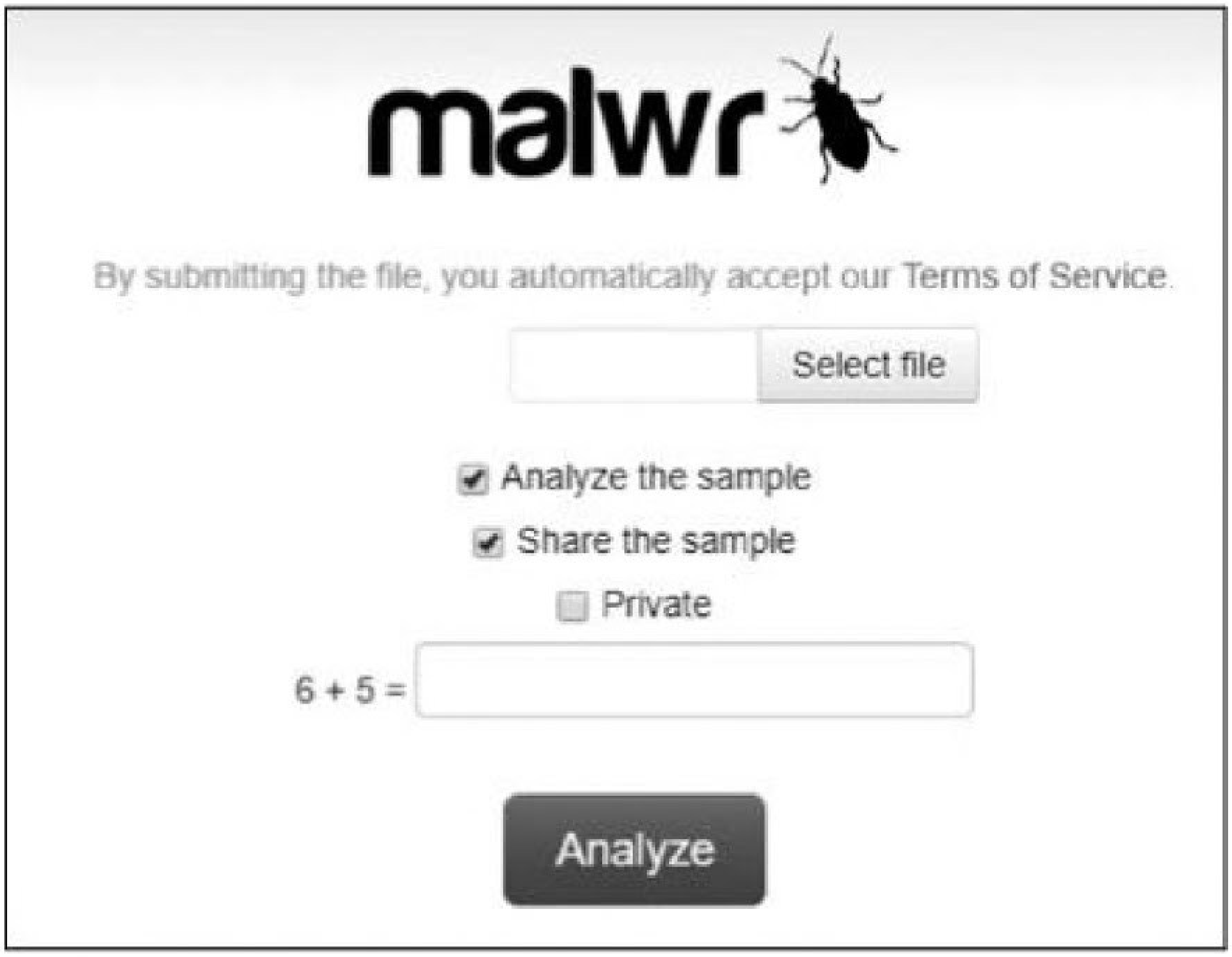
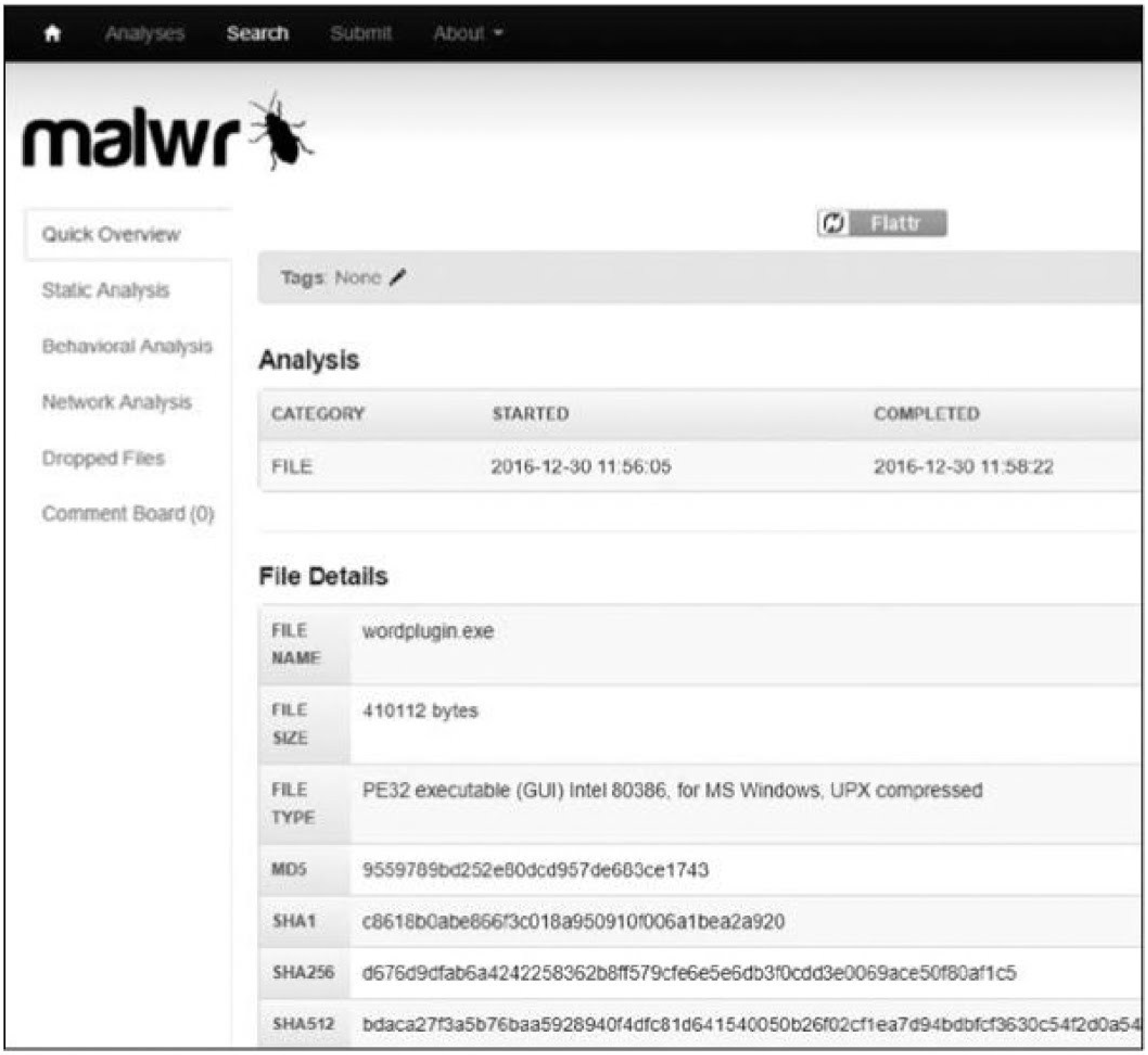


*Figure 3-1: The malware sample submission page*

After you submit your sample through this form, the site should prompt you to wait for analysis to complete, which typically takes about five minutes. When the results load, you can inspect them to understand what the executable did when it was run in the dynamic analysis environment.

### *Analyzing Results on malwr.com*

The results page for our sample should look something like Figure 3-2.



Analysis

| CATEGORY | STARTED | COMPLETED |
|---|---|---|
| FILE | 2016-12-30 11:56:05 | 2016-12-30 11:58:22 |

File Details

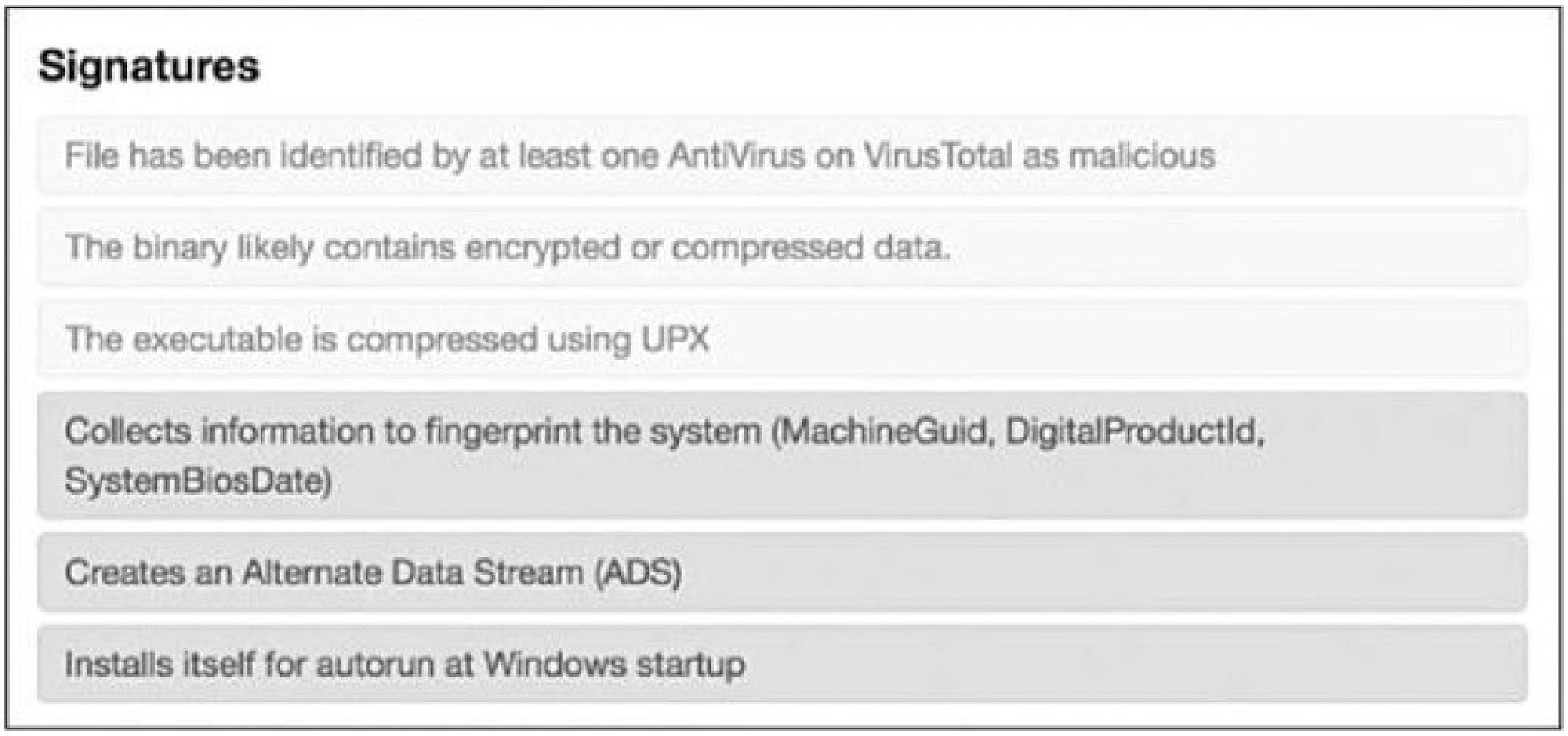
| | |
|---|---|
| FILE NAME | wordplugin.exe |
| FILE SIZE | 410112 bytes |
| FILE TYPE | PE32 executable (GUI) Intel 80386, for MS Windows, UPX compressed |
| MD5 | 9559789bd252e80dcd957de683ce1743 |
| SHA1 | c8618b0abe866f3c018a950910f006a1bea2a920 |
| SHA256 | d676d9dfab6a4242258362b8ff579cfe6e5e6db3f0cdd3e0069ace50f80af1c5 |
| SHA512 | bdaca27f3a5b76baa5928940f4dfc81d641540050b26f02cf1ea7d94bdbfcf3630c54f2d0a54 |

*Figure 3-2: The top of the results page for a malware sample on* malwr.com

The results for this file illustrate some key aspects of dynamic analysis, which we'll explore next.

### Signatures Panel

The first two panels you'll see on the results page are Analysis and File Details. These contain the time the file was run and other static details about the file. The panel I will focus on here is the Signatures panel, shown in Figure 3-3. This panel contains high-level information derived from the file itself and its behavior when it was run in the dynamic analysis environment. Let's discuss what each of these signatures means.


**Signatures**

File has been identified by at least one AntiVirus on VirusTotal as malicious

The binary likely contains encrypted or compressed data.

The executable is compressed using UPX

Collects information to fingerprint the system (MachineGuid, DigitalProductId, SystemBiosDate)

Creates an Alternate Data Stream (ADS)

Installs itself for autorun at Windows startup

*Figure 3-3: The* malwr.com *signatures that match the behavior of our malware sample*

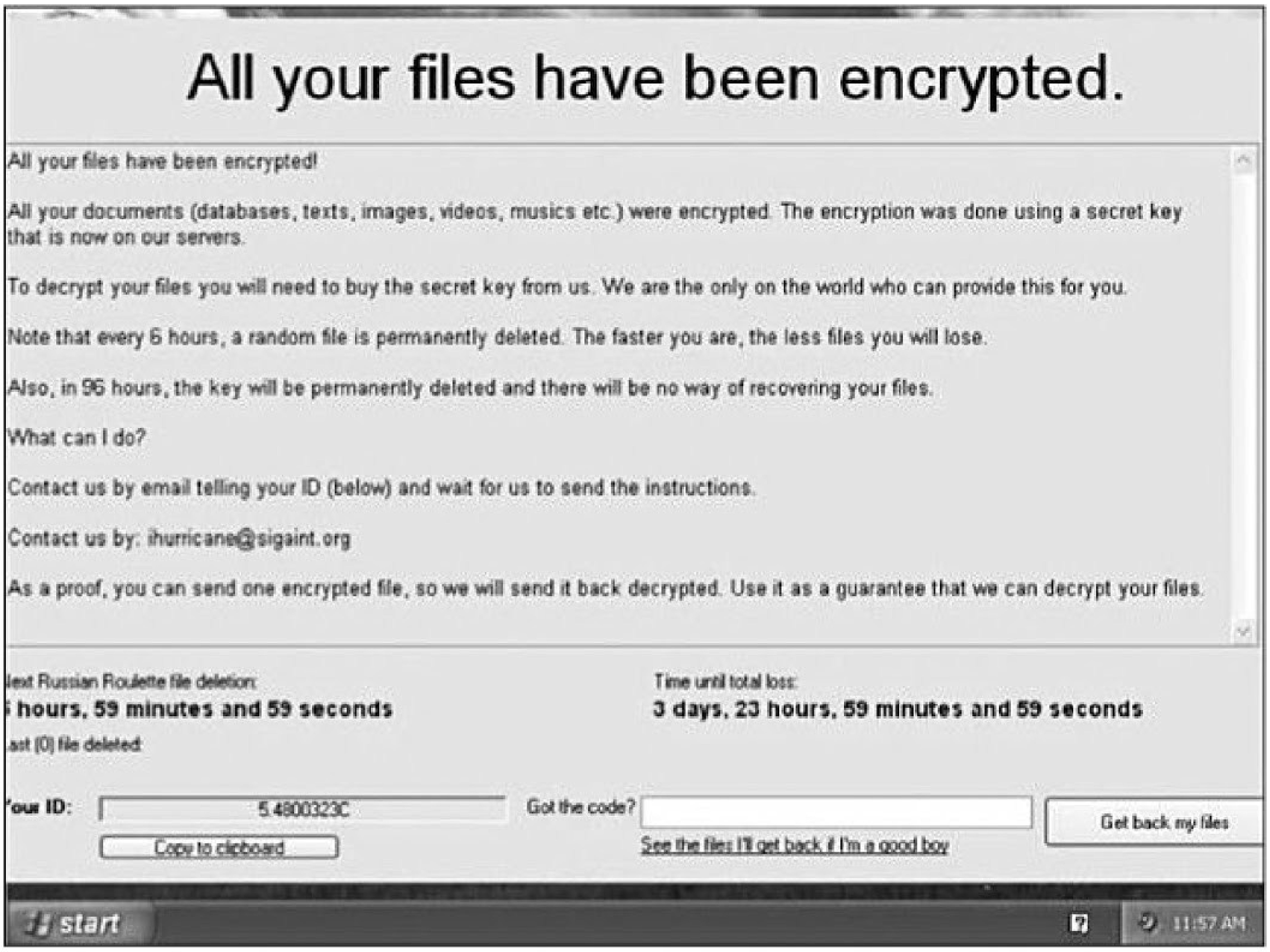
The first three signatures shown in the figure result from static analysis (that is, these are results from the properties of the malware file itself, not its actions). The first signature simply tells us that a number of antivirus engines on the popular antivirus aggregator *VirusTotal.com* marked this file as malware. The second indicates that the binary contains compressed or encrypted data, a common sign of obfuscation. The third tells us that this binary was compressed with the popular UPX packer. Although these static indicators on their own don't tell us what this file does, they do tell us that it's likely malicious. (Note that the color doesn't correspond to static versus dynamic categories; instead, it represents the severity of each rule, with red—the darker gray here—being more suspicious than yellow.)

The next three signatures result from dynamic analysis of the file. The first signature indicates that the program attempts to identify the system's hardware and operating system. The second indicates that the program uses a pernicious feature of Windows known as *Alternate Data Streams (ADS)*, which allows malware to hide data on disk such that it's invisible when using standard file system browsing tools. The third signature indicates that the file changes the Windows registry so that when the system reboots, a program that it specified will automatically execute. This would restart the malware whenever the user reboots their system.

As you can see, even at the level of these automatically triggered signatures, dynamic analysis adds significantly to our knowledge of the file's intended behavior.

## Screenshots Panel

Beneath the Signatures panel is the Screenshots panel. This panel shows a screenshot of the dynamic analysis environment desktop as the malware is running. Figure 3-4 shows an example of what this looks like.



All your files have been encrypted.

All your files have been encrypted!

All your documents (databases, texts, images, videos, musics etc.) were encrypted. The encryption was done using a secret key that is now on our servers.

To decrypt your files you will need to buy the secret key from us. We are the only on the world who can provide this for you.

Note that every 6 hours, a random file is permanently deleted. The faster you are, the less files you will lose.

Also, in 96 hours, the key will be permanently deleted and there will be no way of recovering your files.

What can I do?

Contact us by email telling your ID (below) and wait for us to send the instructions.

Contact us by: ihurricane@sigaint.org

As a proof, you can send one encrypted file, so we will send it back decrypted. Use it as a guarantee that we can decrypt your files.

lext Russian Roulette file deletion:
i hours, 59 minutes and 59 seconds
.ast (0) file deleted:

Time until total loss:
3 days, 23 hours, 59 minutes and 59 seconds

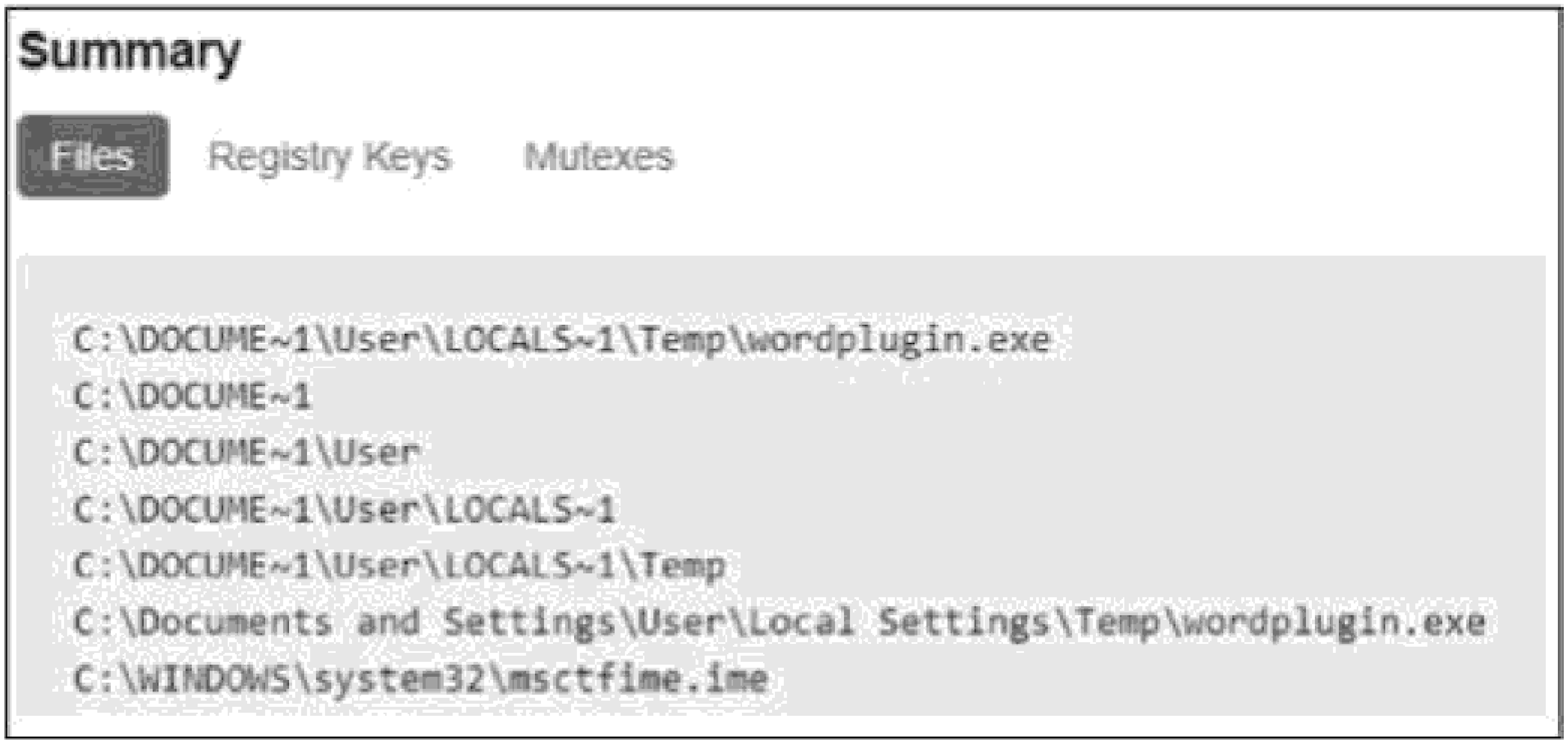
'our ID: 5.4800323C

Figure 3-4: A screen capture of our malware sample's dynamic behavior

You can see that the malware we're dealing with is *ransomware*, which is a type of malware that encrypts a target's files and forces them to pay up if they want to get their data back. By simply running our malware, we were able to uncover its purpose without resorting to reverse engineering.

## Modified System Objects Panel

A row of headings under Screenshots shows the malware sample's network activity. Our binary did not engage in any network communications, but if it had, we would see the hosts it contacted here. Figure 3-5 shows the Summary panel.

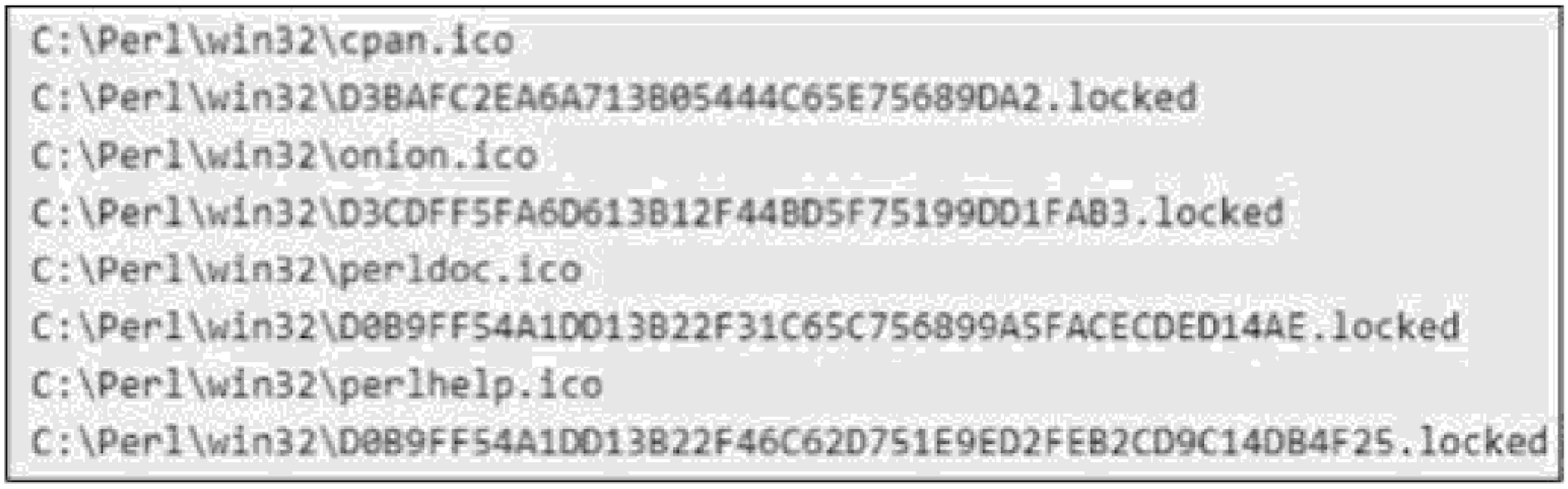


```
C:\DOCUME~1\User\LOCALS~1\Temp\wordplugin.exe
C:\DOCUME~1
C:\DOCUME~1\User
C:\DOCUME~1\User\LOCALS~1
C:\DOCUME~1\User\LOCALS~1\Temp
C:\Documents and Settings\User\Local Settings\Temp\wordplugin.exe
C:\WINDOWS\system32\msctfime.ime
```

*Figure 3-5: The Files tab of the Summary pane, showing which files our malware sample modified*

This shows which system objects, like files, registry keys, and mutexes, the malware has modified.

Looking at the Files tab in Figure 3-6, it's clear that this ransomware malware has indeed encrypted the user files on disk.



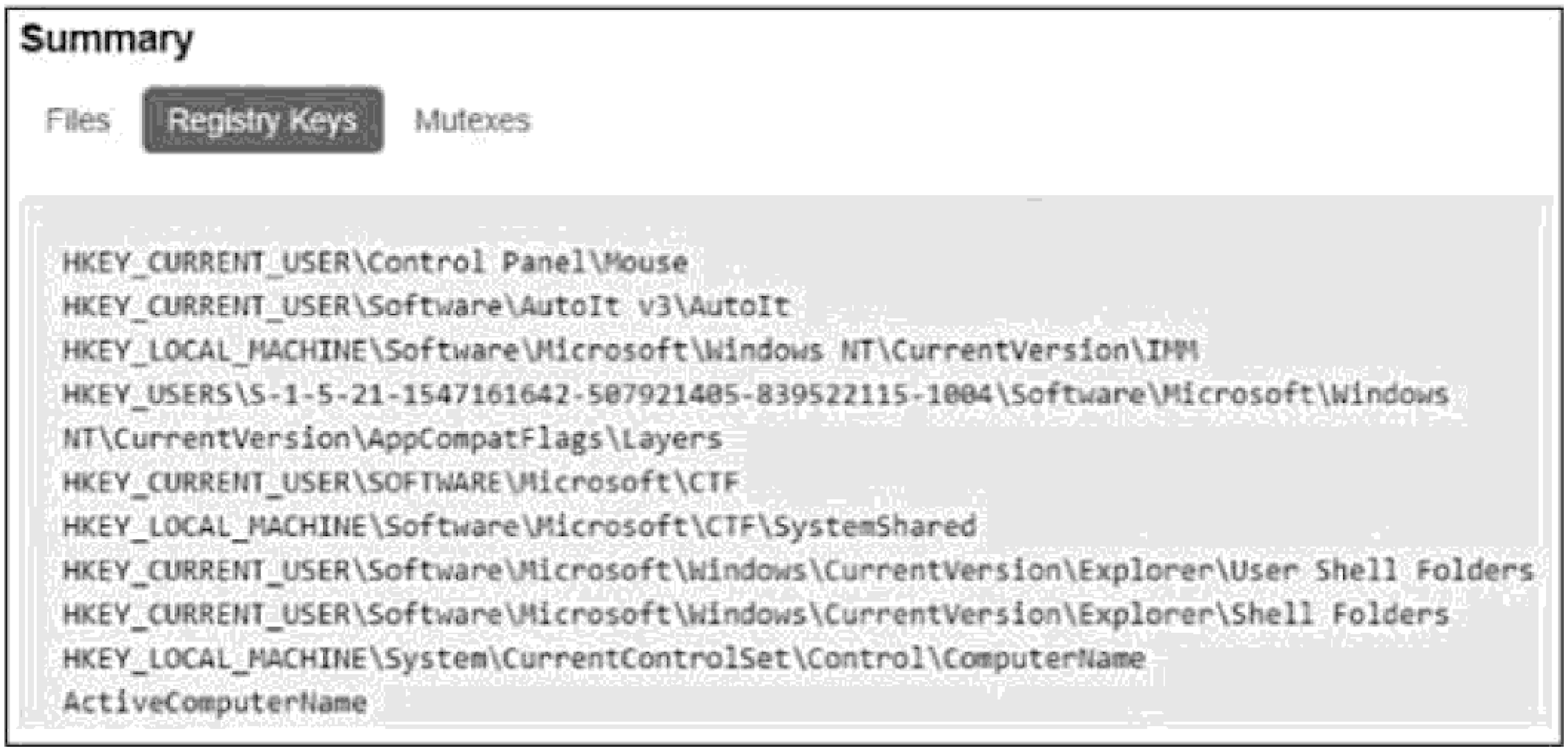
```
C:\Perl\win32\cpan.ico
C:\Perl\win32\D3BAFC2EA6A713B05444C65E75689DA2.locked
C:\Perl\win32\onion.ico
C:\Perl\win32\D3CDFF5FA6D613B12F44BD5F75199DD1FAB3.locked
C:\Perl\win32\perldoc.ico
C:\Perl\win32\D0B9FF54A1DD13B22F31C65C756899A5FACECDED14AE.locked
C:\Perl\win32\perlhelp.ico
C:\Perl\win32\D0B9FF54A1DD13B22F46C62D751E9ED2FEB2CD9C14DB4F25.locked
```

*Figure 3-6: File paths in the Files tab of the Summary pane, suggesting that our sample*

*is ransomware*

After each file path is a file with a *.locked* extension, which we can infer is the encrypted version of the file it has replaced.

Next, we'll look at the Registry Keys tab, shown in Figure 3-7.

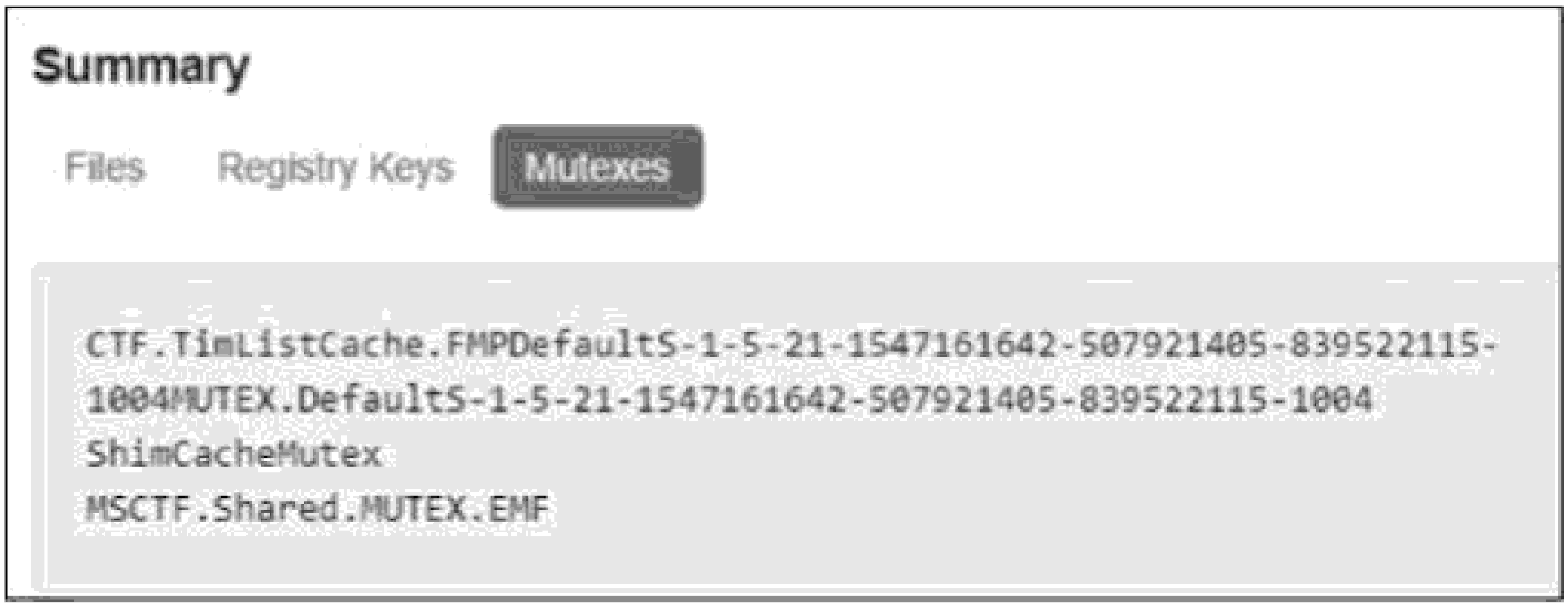


```
HKEY_CURRENT_USER\Control Panel\Mouse
HKEY_CURRENT_USER\Software\AutoIt v3\AutoIt
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\IMM
HKEY_USERS\S-1-5-21-1547161642-507921405-839522115-1004\Software\Microsoft\Windows
NT\CurrentVersion\AppCompatFlags\Layers
HKEY_CURRENT_USER\SOFTWARE\Microsoft\CTF
HKEY_LOCAL_MACHINE\Software\Microsoft\CTF\SystemShared
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ComputerName
ActiveComputerName
```

*Figure 3-7: The Registry Keys tab of the Summary pane, showing which registry keys our malware sample modified*

The registry is a database that Windows uses to store configuration information. Configuration parameters are stored as registry keys, and these keys have associated values. Similar to file paths on the Windows file system, registry keys are backslash delimited. *Malwr.com* shows us what registry keys our malware modified. Although this isn't shown in Figure 3-7, if you view the complete report on *malwr.com*, you should see that one notable registry key our malware changed is `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`, which is a registry key that tells Windows to run programs each time a user logs on. It's very likely that our malware modifies this registry to tell Windows to restart the malware every time the system boots up, which ensures that the malware infection persists from reboot to reboot.

The Mutexes tab in the *malwr.com* report contains the names of the mutexes the malware created, as shown in Figure 3-8.



```
CTF.TimListCache.FMPDefaultS-1-5-21-1547161642-507921405-839522115-
1004MUTEX.DefaultS-1-5-21-1547161642-507921405-839522115-1004
ShimCacheMutex
MSCTF.Shared.MUTEX.EMF
```

*Figure 3-8: The Mutexes tab of the Summary pane, showing which mutexes our malware sample created*

Mutexes are lock files that signal that a program has taken possession of some resource. Malware often uses mutexes to prevent itself from infecting a system twice. It turns out that at least one mutex created (*CTF.TimListCache.FMPDefaultS-1-5-21-1547161642-507921405-839522115-1004MUTEX.DefaultS-1-5-21-1547161642-507921405-839522115-1004 ShimCacheMutex*) is known by the security community to be associated with malware and may be serving this purpose here.

## API Call Analysis

Clicking the Behavioral Analysis tab on the left panel of the *malwr.com* UI, as shown in Figure 3-9, should bring up detailed information about our malware binary's behavior.

This shows what API calls were made by each process launched by the malware, along with their arguments and return values. Perusing this information is time consuming and requires expert knowledge of Windows APIs. Although a detailed discussion of malware API call analysis is beyond the scope of this book, if you're interested in learning more, you can look up individual API calls to discover their effects.

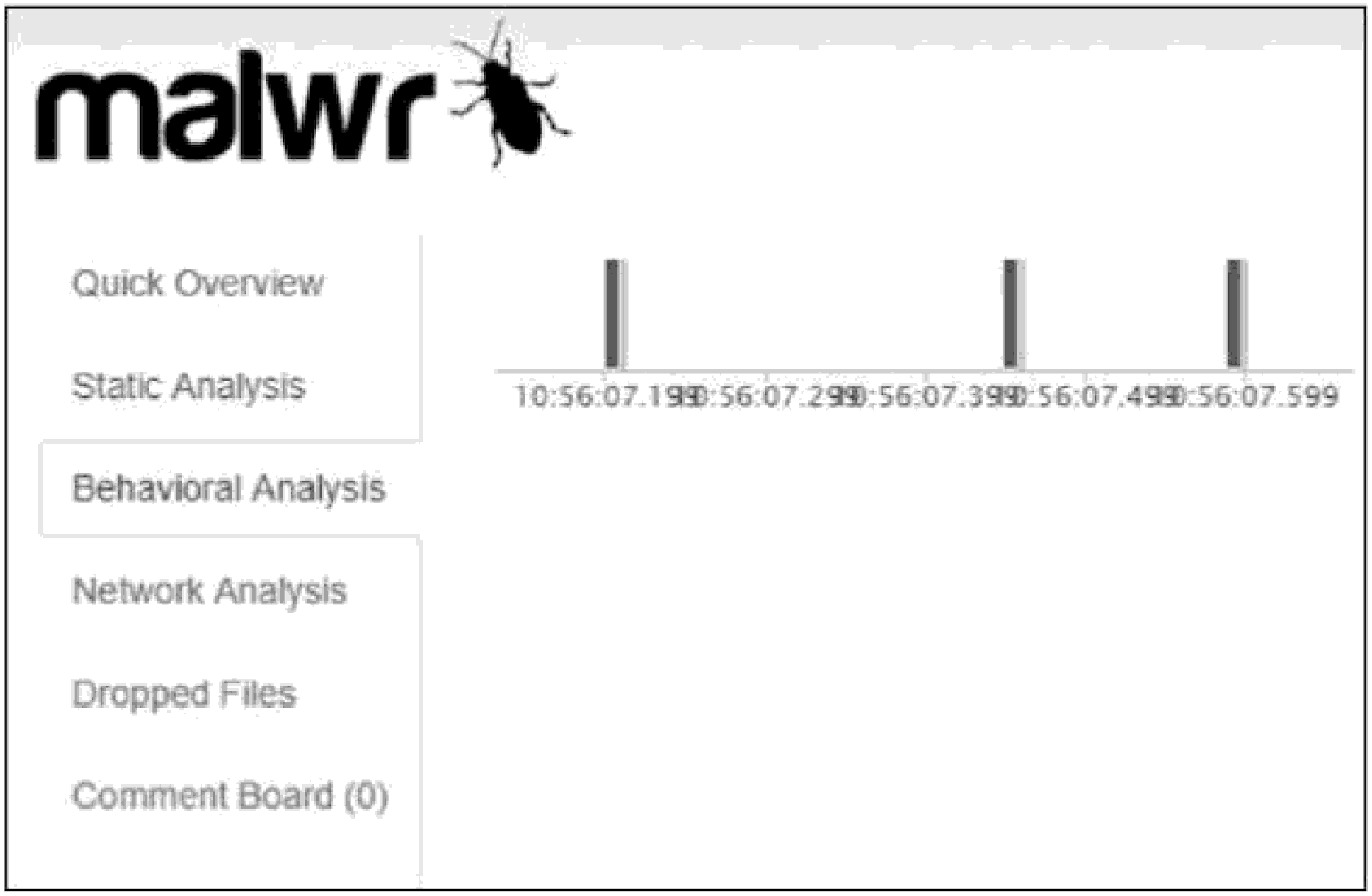


*Figure 3-9: The Behavioral Analysis pane of the* malwr.com *report for our malware sample, showing when API calls were made during the dynamic execution*

Although *malwr.com* is a great resource for dynamically analyzing individual malware samples, it isn't great for performing dynamic analysis on large numbers of samples. Executing large numbers of samples in a dynamic environment is important for machine learning and data analysis because it identifies relationships between malware samples' dynamic execution patterns. Creating machine learning systems that can detect instances of malware based on their dynamic execution patterns requires running thousands of malware samples.

In addition to this limitation, *malwr.com* doesn't provide malware analysis results in machine-parseable formats like XML or JSON. To address these issues you must set up and run your own CuckooBox. Fortunately, CuckooBox is free and open source. It also comes with step-by-step instructions for setting up your very own dynamic analysis environment. I encourage you to do so by going to *http://cuckoosandbox.org/*. Now that you understand how to interpret dynamic malware results from *malwr.com*, which uses CuckooBox behind the scenes, you'll also know how to analyze CuckooBox results once you have CuckooBox up and running.

## Limitations of Basic Dynamic Analysis

Dynamic analysis is a powerful tool, but it is no malware analysis panacea. In fact, it has serious limitations. One limitation is that malware authors are aware of CuckooBox and other dynamic analysis frameworks and attempt to circumvent them by making their malware fail to execute when it detects that it's running in CuckooBox. The CuckooBox maintainers are aware that malware authors try to do this, so they try to get around attempts by malware to circumvent CuckooBox. This cat-and-mouse game plays out continuously such that some malware samples will inevitably detect that they are running in dynamic analysis environments and fail to execute when we try to run them.

Another limitation is that even without any circumvention attempts, dynamic analysis might not reveal important malware behaviors. Consider the case of a malware binary that connects back to a remote server upon execution and waits for commands to be issued. These commands may, for example, tell the malware sample to look for certain kinds of files on the victim host, to log keystrokes, or turn on the webcam. In this case, if the remote server sends no commands, or is no longer up, none of these malicious behaviors will be revealed. Because of these limitations, dynamic analysis is not a fix-all for malware analysis. In fact, professional malware analysts combine dynamic and static analysis to achieve the best possible results.

## Summary

In this chapter you ran dynamic analysis on a ransomware malware sample with *malwr.com* to analyze the results. You also learned about the advantages and shortcomings of dynamic analysis. Now that you've learned how to conduct basic dynamic analysis, you're ready to dive into malware data science.

The remainder of this book focuses on performing malware data science on static analysis–based malware data. I'll focus on static analysis because it's simpler and easier to get good results with compared to dynamic analysis, making it a good starting place for getting your hands dirty with malware data science. However, in each subsequent chapter I'll also explain how you can apply data science methods to dynamic analysis–based data.

# 4
# IDENTIFYING ATTACK CAMPAIGNS USING MALWARE NETWORKS

*Malware network analysis* can turn malware datasets into valuable threat intelligence, revealing adversarial attack campaigns, common malware tactics, and sources of malware samples. This approach consists of analyzing the ways in which groups of malware samples are connected by their shared attributes, whether those are embedded IP addresses, hostnames, strings of printable characters, graphics, or similar.

For example, Figure 4-1 shows an example of the power of malware network analysis in a chart that took only seconds to generate with the techniques you'll learn in this chapter.

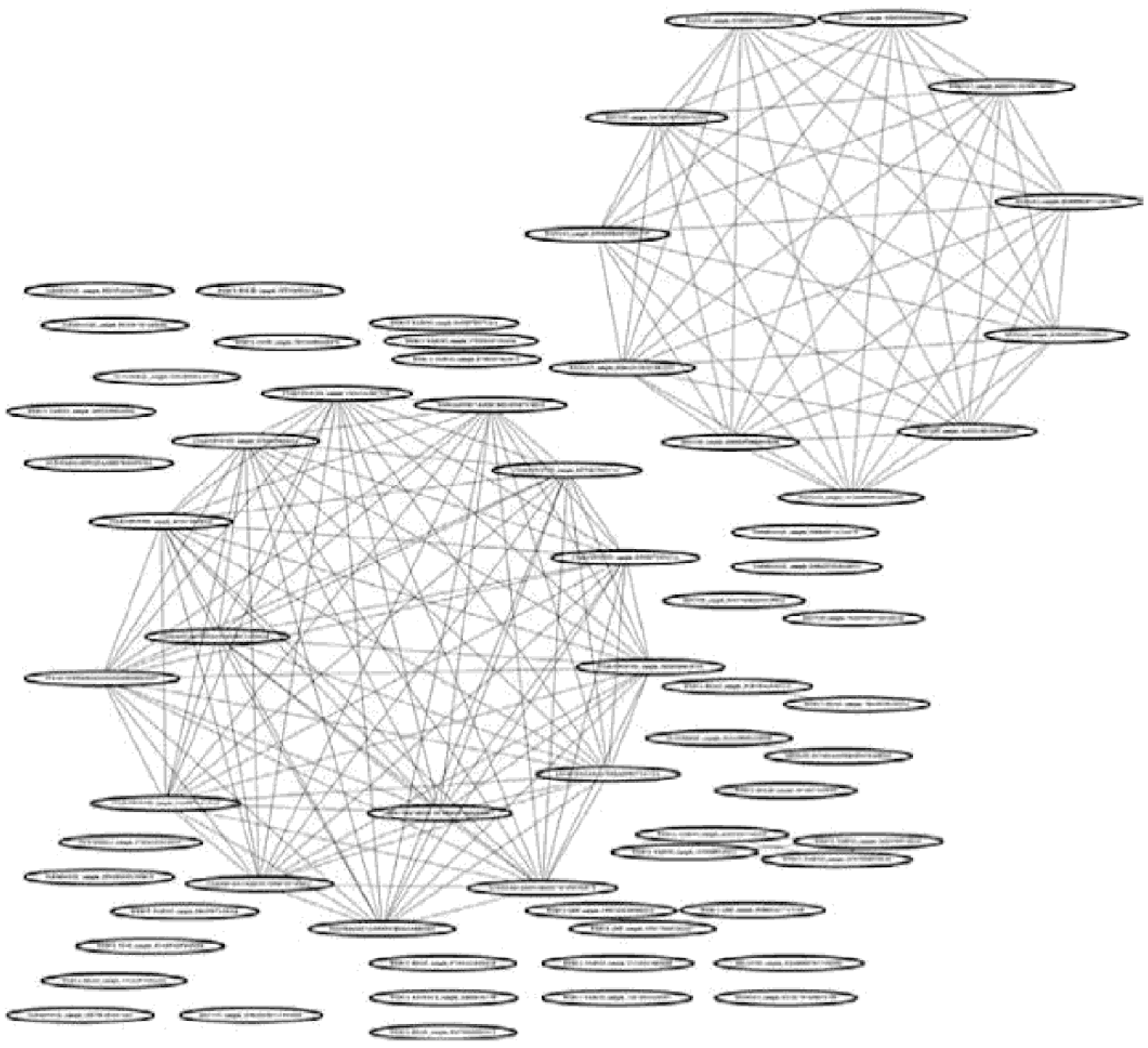


*Figure 4-1: Nation-state malware's social network connections revealed via shared attribute analysis*

The figure displays a group of nation state–grade malware samples (represented as oval-shaped nodes) and their "social" interconnections (the lines connecting the nodes). The connections are based on the fact that these samples "call back" to the same hostnames and IP addresses, indicating they were deployed by the same attackers. As you'll learn in this chapter, you can use these connections to help differentiate between a coordinated attack on your organization and a disparate array of criminally motivated attackers.

By the end of the chapter you will have learned:

- The fundamentals of network analysis theory as it relates to extracting threat intelligence from malware
- Ways to use visualizations to identify relationships between malware samples
- How to create, visualize, and extract intelligence from malware networks using Python and various open source toolkits for data analysis and visualization
- How to tie all this knowledge together to reveal and analyze attack campaigns within real-world malware datasets

## Nodes and Edges

Before you can perform shared attribute analysis on malware, you need to understand some basics about networks. *Networks* are collections of connected objects (called *nodes*). The connections between these nodes are referred to as *edges*. As abstract mathematical objects, the nodes in a network can represent pretty much anything, as can their edges. What we care about for our purposes is the structure of the interconnections between these nodes and edges, as this can reveal telling details about malware.

When using networks to analyze malware, we can treat each individual malware file as the definition of a node, and we can treat relationships of interest (such as shared code or network behavior) as the definition of an edge. Similar malware files share edges and thus cluster together when we apply force-directed networks (you will see exactly how this works later). Alternatively, we can treat both malware samples and attributes as nodes unto themselves. For example, callback IP addresses have nodes, and so do malware samples. Whenever malware samples call back to a particular IP address, they are connected to that IP address node.

Networks of malware can be more complex than simply a set of nodes and edges. Specifically, they can have *attributes* attached to either nodes or edges, such as the percentage of code that two connected samples share. One common edge attribute is a *weight*, with greater weights indicating stronger connections between samples. Nodes may have their own attributes, such as the file size of the malware samples they represent, but these are typically referred to only as attributes.

## Bipartite Networks

A *bipartite network* is one whose nodes can be divided into two partitions (groups), where neither partition contains internal connections. Networks of this type can be used to show shared attributes between malware samples.

Figure 4-2 shows an example of a bipartite network in which malware sample nodes go in the bottom partition, and domain names the samples “call back” to (in order to communicate with the attacker) go in the other partition. Note that callbacks never connect directly to other callbacks, and malware samples never connect directly to other malware samples, as is characteristic of a bipartite network.

As you can see, even such a simple visualization reveals an important piece of intelligence: based on the malware samples’ shared callback servers, we can guess that *sample_014* was probably deployed by the same attacker as *sample_37D*. We can also guess that *sample_37D* and *sample_F7F* probably share the same attacker, and that *sample_014* and *sample_F7F* probably share the same attacker, because they’re connected by sample *sample_37D* (and indeed, the samples shown in Figure 4-2 all come from the same “APT1” Chinese attacker group).

**NOTE**

*We’d like to thank Mandiant and Mila Parkour for curating the APT1 samples and making them available to the research community.*

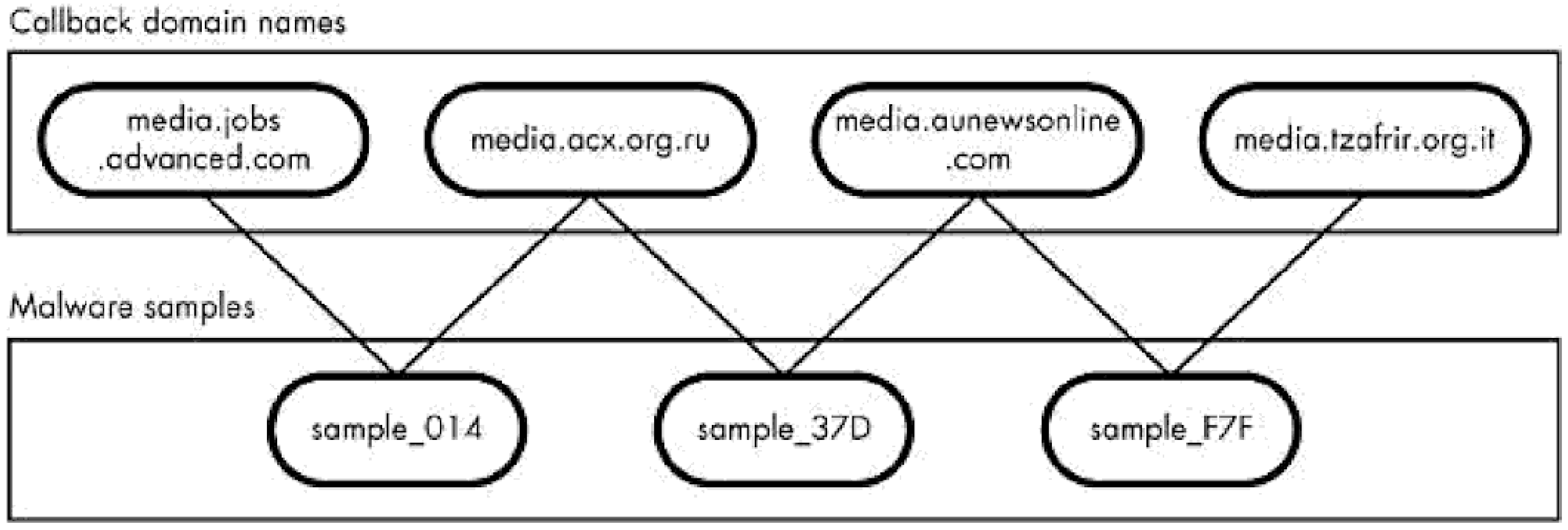


*Figure 4-2: A bipartite network. The nodes on top (the attributed partition) are callback domain names. The nodes on the bottom (malware partition) are malware samples.*

As the number of nodes and connections in our network grow very large, we might want to see just how the malware samples are related, without having to closely inspect all the attribute connections. We can examine malware sample similarity by creating a bipartite network *projection*, which is a simpler version of a bipartite network in which we link nodes in one partition of the network if they have nodes in the other partition (the *attribute* partition) in common. For example, in the case of the malware samples shown in Figure 4-1, we'd be creating a network in which malware samples are linked if they share callback domain names.

Figure 4-3 shows the projected network of the shared-callback servers of the entire Chinese APT1 dataset referred to previously.

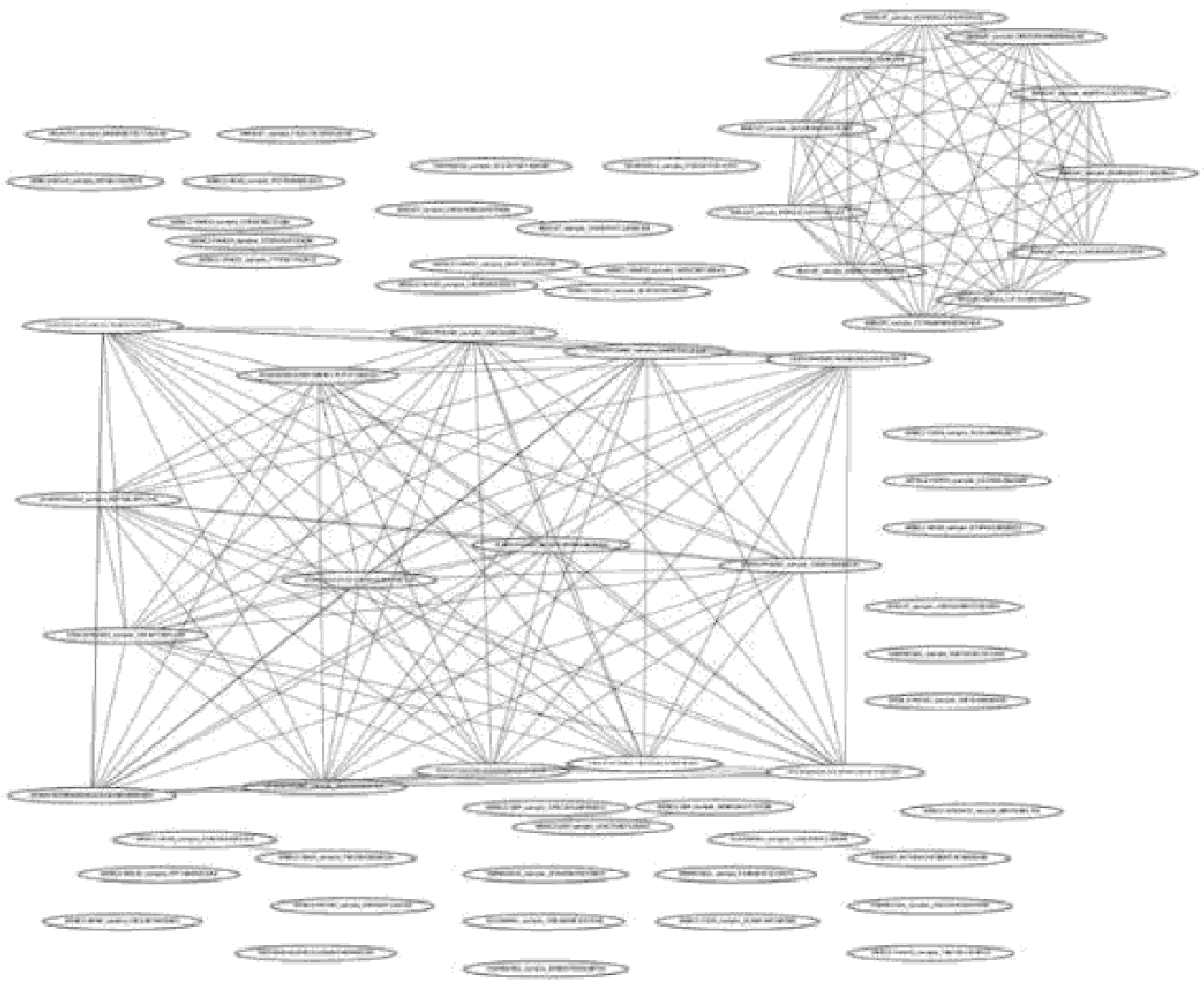


*Figure 4-3: A projection of malware samples from the APT1 dataset, showing connections between malware samples only if they share at least one server. The two big clusters were used in two different attack campaigns.*

The nodes here are malware samples, and they are linked if they share at least one callback server. By showing connections between malware samples only if they share callback servers, we can begin to see the overall "social network" of these malware samples. As you can see in Figure 4-3, two large groupings exist (the large square cluster in the left-center area and the circular cluster in the top-right area), which upon further inspection turn out to correspond to two different campaigns carried out over the APT1 group's 10-year history.

## Visualizing Malware Networks

As you perform shared attribute analysis of malware using networks, you'll find that you rely heavily on network visualization software to create the networks like the ones shown thus far. This section introduces how these network visualizations can be created from an algorithmic perspective.

Crucially, the major challenge in doing network visualization is *network layout*, which is the process of deciding where to render each node in a network within a two- or three-dimensional coordinate space, depending on whether you want your visualization to be two- or three-dimensional. When you're placing nodes on a network, the ideal way is to place them in the coordinate space such that their visual distance from one another is proportional to the shortest-path distance between them in the network. In other words, nodes that are two hops away from one another might be about two inches away from one another, and nodes that are three hops away might be about three inches apart. Doing this allows us to visualize clusters of similar nodes accurately to their actual relationship. As you'll see in the next section, however, this is often difficult to achieve, especially when you're working with more than three nodes.

### *The Distortion Problem*

As it turns out, it's often impossible to solve this network layout problem perfectly. Figure 4-4 illustrates this difficulty.

As you can see in these simple networks, all nodes are connected to all other nodes by edges of equal weights of 1. The ideal layout for these connections would place all nodes equidistant from one another on the page. But as you can see, as we create networks of four and then five nodes, as in (c) and (d), we start to introduce progressively more distortion due to edges of unequal length. Unfortunately, we can only minimize, not eliminate this distortion, and that minimization becomes one of the major goals of network visualization algorithms.