UNIVERSITY OF CALIFORNIA

Santa Barbara

Defending Against Malicious Software

A Dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Robert Bradley Gilbert

Committee in charge:

Professor Richard Kemmerer, Chair

Professor Christopher Kruegel

Professor Giovanni Vigna

December 2011

The dissertation of Robert Bradley Gilbert is approved.

Professor Giovanni Vigna

Professor Christopher Kruegel

Professor Richard Kemmerer, Chair

September 2011

Defending Against Malicious Software

by

Robert Bradley Gilbert

DEDICATION

I dedicate this dissertation to my mother, Pam, and my father, Brad.

My mother has taught me the value of perseverance and diligence. Her determination has inspired my own, and her ever-present love will always be felt, wherever my path takes me.

My father has worked harder than any man I have known. For decades, he has moved the earth and fished the sea so that I could work comfortably in a digital world. He is a selfless man who has taught me to be devoted and honorable in all that I do, and I will never forget it.

ACKNOWLEDGMENTS

1001 people have inspired me, in one way or another, to pursue this degree and complete this dissertation. I would like to acknowledge just a few of them here.

I want to thank my advisor, Dick Kemmerer, whose guidance has been invaluable over the years. He was always willing to offer his assistance, all the while challenging me to think critically and independently. Dick has invariably looked out for my best interests, concerning my life both within and outside of the lab. For this, I am eternally grateful.

I also wish to thank the other members of my committee that I've had the great fortune of working with: Chris Kruegel and Giovanni Vigna. These men were instrumental in helping me incubate the ideas that were brought to fruition in this dissertation. I owe a special debt of gratitude to them both. I could not have done this without them.

I would like to acknowledge all of the members of the Seclab whom I have been lucky enough to work along side. We have cultivated a special research group here that facilitates each other intellectually while encouraging each other socially. I will never forget the paper deadlines, the reading groups, or the Gaucho Ball. I thank them all for the late nights, the early mornings, and everything in between.

Fortunate does not begin to describe the experience of having the family that I do. I would not be here today without the constant and unwaivering support of my mother and father, stepparents, godparents, siblings, and, of course, grandmates. I thank them, truly, from the bottom of my heart.

In particular, I want to acknowledge my grandfather, Orville, for teaching me that my only limitation is my imagination. He is a student of technology, a loving patriarch, and a great man.

Finally, I thank Betti for remaining so close to me, even when she is 6000 miles away.

VITA OF ROBERT BRADLEY GILBERT

September 2011

EDUCATION

University of California, Santa Barbara
Ph.D., Computer Science
Santa Barbara, CA USA
September 2004 – December 2011

University of California, Santa Barbara
M.S., Computer Science
Santa Barbara, CA USA
September 2004 – March 2011

University of San Diego
B.A., Computer Science
Summa Cum Laude
Phi Beta Kappa Honor Society
San Diego, CA USA
September 2000 – May 2004

ACADEMIC EXPERIENCE

University of California, Santa Barbara
Research Assistant, Computer Security Group
Santa Barbara, CA USA
September 2006 – September 2011
Advised by Professor Richard Kemmerer

PROFESSIONAL EXPERIENCE

Microsoft Corporation
Software Design Engineer Intern
Redmond, WA USA
June 2009 – September 2009

- Joined adCenter Delivery Engine Fraud team
- Developed software and models to improve bot clickfraud detection

Microsoft Corporation
Software Design Engineer Intern
Redmond, WA USA
July 2008 – September 2008

- Rejoined Internet Crime Investigations team
- Analyzed the effectiveness of HTTP as a botnet C&C channel
- Presented work at 8th International Botnet Task Force conference

Microsoft Corporation
Software Design Engineer Intern
Redmond, WA USA
July 2007 – September 2007

- Joined Internet Crime Investigations team
- Developed an application to track and monitor malware infection
- Featured in New York Times article: "A Robot Network Seeks to Enlist Your Computer" by John Markoff, 10/21/08

PROFESSIONAL EXPERIENCE (CONT.)

Aspera, Inc.
Software Engineering Intern
Berkeley, CA USA
June 2006 – September 2006

- Implemented API to manage Aspera Fasp protocol security
- Added provisions for Fasp transfered files to remain encrypted on disk, only to be opened by authorized recipients

Aspera, Inc.
Software Engineering Intern
Berkeley, CA USA
June 2005 – September 2005

- Implemented API to manage Aspera Scp file transfer jobs
- Integrated Aspera Scp file transfer into Avid systems for the Canadian Broadcasting Corporation (CBC) to use for the 2006 Winter and 2008 Summer Olympic Games

ABSTRACT

Defending Against Malicious Software

by

Robert Bradley Gilbert

Contemporary malicious software is crafted and deployed by sophisticated and highly organized criminal enterprises. Modern threats include advanced spyware and large-scale botnets that are designed to profit from the vulnerabilities that are inherent in the well-connected, Internet-enabled computing ecosystem. Furthermore, nation-states have begun leveraging malware in targeted attacks against their enemies to steal state secrets or even damage physical infrastructure.

A considerable security community has formed to combat the ever-evolving malware threat, inspiring an arms race between attackers and defenders. To wit, malware authors continue to improve their techniques for the delivery and concealment of malicious payloads. Meanwhile, researchers have enhanced their approaches to defense, which can be generally divided into three stages: analysis, detection, and response. While valuable work has been done in each of these areas, malware, unfortunately, still flourishes.

In this dissertation, we take a holistic approach to improving malware defense by making novel contributions to all three of the aforementioned stages. In particular, we discuss our analysis of the Torpig botnet, in which we obtained unique insights into the operations of the malicious network by taking control of it for a period of ten days. Our analysis demonstrates how sophisticated and destructive contemporary malware has become. This motivated our development of two host-based systems to detect and contain such malware. The first approach detects malware by implementing a dynamic code identity primitive. The second system contains malware by blocking malicious attempts to interact with trusted processes to carry out hostile actions. Collectively, these systems offer an effective and complementary approach to mitigating the threat of advanced malware.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

In 1982, a ninth grade student named Rich Skrenta created the first known computer virus to target the commodity personal computer. His "Elk Cloner" program was devised to be a practical joke [179], as its effect was to display a poem on the terminal display upon every 50th boot of an infected floppy disk:

> *Elk Cloner: The program with a personality*
>
> *It will get on all your disks*
>
> *It will infiltrate your chips*
>
> *Yes it's Cloner!*
>
> *It will stick to you like glue*
>
> *It will modify ram too*
>
> *Send in the Cloner!*

In June 2010, the Stuxnet worm was discovered to be one of the most complex and dangerous examples of malicious software ever created [41]. Stuxnet comprised a targeted attack against Supervisory Control and Data Acquisition (SCADA) systems that controlled a very specific programmable logic controller (PLC) type and configuration. It seems that the goal of the attack was to reprogram a particular industrial control system, allegedly one controlling the Natanz nuclear enrichment facility in Iran [118]. Upon analysis of the software, a prominent computer security company portended that Stuxnet is "a working and fearsome prototype of a cyber-weapon that will lead to the creation of a new arms race in the world" [88].

Clearly, malicious software (colloquially known as *malware*) has evolved considerably, both in intent and technical sophistication, over the last 30 years. In the early days of malware technology, computer viruses and worms were created as experiments or mere annoyances [189]. Today, the threat is much more dangerous as, for example, profit-seeking criminal enterprises are using malware to join infected computers into large-scale botnets in order to steal and monetize the personal information of their victims [163, 69]. Furthermore, as the Stuxnet malware illustrates, today's threats include advanced targeted attacks with implications that nation-states are leveraging malware to engage in cyber-warfare [194, 114, 115, 117, 116].

There are two contributing factors that have driven the motivation of malware authors to create ever more insidious threats. First, the Microsoft Windows operating system has become incredibly popular for personal computer use, boasting 92.90% of the OS market share as of September 2011, according to one survey [139]. This software monoculture means that a malicious program that targets the Windows platform will

run on a large fraction of the world's computer resources, effectively amplifying the impact of a single malware deployment. Second, computer users are more vulnerable to attack because they are well-connected, due to the widespread adoption of the Internet and, in particular, its primary service, the World Wide Web.

The web itself has transformed significantly from its original intended purpose. In 1991, engineers at CERN created the World Wide Web in order to facilitate the sharing of data among a small group of scientific collaborators [13]. Soon after, the incredible potential of the web as a means for information dissemination was realized, and it grew quickly, both in size and popularity. Today, in addition to enabling all of its users to publish simple static pages or rich dynamic content, the web also provides a platform upon which full-featured applications and services are built for consumption by an ever growing community. It is estimated that the number of web users eclipses 2.10 billion, which is about 30.23% of the world population [131].

The rise of the web has revolutionized the way its many users interact, but it has also provided the opportunity for miscreants to do significant harm. In particular, there are three primary characteristics of the modern web that draw malicious attention. First, as the web has become nearly ubiquitous, users today commit exceedingly sensitive information to their computers (and to the web itself) to support their use of web applications such as email, electronic commerce, online banking, and social networking. Second, these and other web applications and services are very popular, realizing millions of users each. Finally, the applications that are hosted on the web are vulnerable to attack. This last aspect of the web is illustrated in Table 1.1, which presents web-related security vulnerabilities recorded in the Common Vulnerabilities and Exposures

(CVE) database. Table 1.1 demonstrates that reported vulnerabilities in all software have grown in the last decade and that the proportion of vulnerabilities in web-related technologies, such as servers, web applications, and client web browsers, is significant.

Table 1.1: Web-related vulnerabilities per year [191].

Year	Web-related	Total	Proportion
1999	159	1,573	10.11%
2000	222	1,235	17.98%
2001	407	1,566	25.99%
2002	829	2,422	34.23%
2003	441	1,591	27.72%
2004	983	2,769	35.50%
2005	2,347	4,887	48.03%
2006	4,539	7,245	62.65%
2007	3,668	6,743	54.40%
2008	4,329	7,290	59.38%
2009	2,637	5,082	51.89%
2010	2,082	4,806	43.32%
Total	22,643	47,209	47.96%

Attackers exploit vulnerabilities in software as a means toward a wide variety of illicit ends [48], and malware is commonly deployed during exploitation in order to reach these goals. As one could imagine, along with the rise of discovered software vulnerabilities, there is an incidental increase in the amount of distributed malware. Table 1.2 shows the number of new malware signatures created each year by a prominent security software company in order to keep pace with the growing malware threat. A malware signature is written in order to statically identify a known malware instance. The number of signatures in recent years has grown dramatically (the total in 2009 translates to one signature written every 11 seconds), but this is largely because malware authors are

using advanced obfuscation techniques, such as emulation technology [174] and executable compression [66], to make each malware instance seem unique. Therefore, the numbers do not necessarily indicate a one-to-one correspondence between a malware signature and a distinct malware functionality. Nonetheless, the data demonstrates that malware authors are highly skilled, and malware defense practitioners must develop increasingly advanced techniques of their own to cope with the threat.

Table 1.2: Symantec new malware signatures per year [49].

Year	**Signatures**
2002	20,254
2003	19,159
2004	74,981
2005	113,081
2006	167,069
2007	708,742
2008	1,691,323
2009	2,895,802
Total	5,690,411

Obviously, the malware problem is far from solved. While research is ongoing in the areas of vulnerability analysis [150], detection [80], and prevention [97], the focus of this dissertation is not on impeding the vulnerabilities that allow malware to proliferate, but rather on proposing an array of defense techniques to thwart the malware itself. Before addressing the approaches to malware defense in Section 1.2, it is advantageous to first identify and classify the various malware technologies in order to more fully understand the threat.

1.1 Classes of Malware

The names and definitions of different types of malware are often used interchangeably or incorrectly. This leads to confusion about the nature of malware and makes it difficult to reason about the intentions, effects, and capabilities of a given malware instance. The problem is getting the attention of the security community, and a standardized language for characterizing malware is currently under development [98].

This section presents a classification of the various malware types. The categorization arranges malware types into three distinct classes: *delivery*, *concealment*, and *payload*. While a given malware instance may consist of components from multiple classes (in particular, a malware payload requires a delivery component), it is helpful to describe the components individually, as defensive techniques are often employed to target these more specific aspects of malware individually.

1.1.1 Malware as a Delivery Vehicle

In order for malware to have any sort of effect, it must first find a way to breach the target system. Malware authors utilize a number of techniques to achieve this goal, both through technical means and social engineering efforts. While the methods of exploiting victims vary, the purpose is the same: to deliver the malware to the target computer and execute it. The various malware delivery techniques are now described.

Computer Virus

A computer virus is a program that is designed to replicate itself by modifying other programs in order to infect them with a copy of itself [25]. When a contaminated program is subsequently executed, the attached virus is also executed, and it infects still other programs, thus propagating the malware in a manner similar to how viruses proliferate in the natural world. A virus spreads to other computers when a user copies an infected program to a remote system, via removable media (e.g., a floppy disk or USB flash drive) or over the network. Viruses have one of three primary infection strategies: *boot sector infection*, *executable infection*, or *macro infection*.

Boot Sector Infection. Boot sector infecting viruses either replace portions of code in the boot loader program in the master boot record (MBR) of a hard disk, or they modify the boot sector code in the volume boot record (VBR) of a floppy disk or hard disk partition. When an infected disk (either the hard drive or a floppy) is booted, the virus code is executed. Typically, the virus stays resident in memory and monitors the system so that it can infect the boot sector of other (floppy) disks when they are booted.

Executable Infection. Executable file infecting viruses propagate by inserting the virus code into other executable files on disk. The most straightforward approach taken by these viruses is to simply overwrite the host program with the virus code. This has the side effect of damaging the original file, which brings attention to the virus' existence and hinders its propagation.

A more advanced technique, a *parasitic virus*, overcomes this limitation by appending the virus code to the host program and updating the program entry point so that execution begins at the virus before it passes control to the host program. A parasitic virus changes the size of the host file, possibly betraying its presence, so other techniques have been used for further obfuscation. A *cavity virus* is a type of parasitic virus that injects code into unused regions of the host program's code segment, so that the file size of the host executable remains the same.

Since parasitic viruses typically modify the program entry point in order to get their code to execute, it is relatively easy for fast virus scanning tools to discover their presence by scanning code around the entry point. Virus authors have responded by using *entry point obfuscation* techniques, which leave the program entry point unchanged and instead overwrite Import Address Table (IAT) addresses or function call instructions to pass control to the virus code [188].

Further advanced techniques have been developed to thwart virus detection. Virus authors utilize *code integration* techniques to interleave virus code with the original host program in order to make it difficult to differentiate the two [42]. A *polymorphic virus* changes its appearance with each infection by encrypting the virus code with a different key for each instance. Finally, a *metamorphic virus* produces code for each virus instance that consists of different individual instructions, but the code for all instances is semantically the same.

Macro Infection. Unlike boot sector and executable infecting viruses, macro viruses infect data files. These viruses became popular when modern applications, such as

the Microsoft Office suite of tools, began including powerful macro languages to support advanced automation features. At one time, macro infecting viruses were quite effective at spreading because most applications were configured to allow macros to automatically run when their host file was loaded. Once executed, the virus code will typically copy itself to other data files on the system and perhaps facilitate its spread by making use of other capabilities of the macro language, such as composing emails and attaching infected files before sending them [53].

Computer Worm

A computer worm is a self-contained program that propagates over the network to remote hosts, either autonomously or with the assistance of its unwitting victims.[1] After gaining access to a remote system, a worm typically enters a target acquisition phase in which it locates additional victims. The worm is then sent to these targets, and the cycle repeats. Worms can be divided into two categories, based on their method of propagation: *exploit-based* and *email-based*.

Exploit-Based Worm. Exploit-based worms are designed to exploit security vulnerabilities in well-known network-based services [206]. Worms in this class tend to spread very fast because they require no human interaction to propagate. In fact, the propagation speed of these worms is typically limited only by network latency [134] (when using TCP as the transport protocol) or available bandwidth [133] (when us-

[1] Note that self-containment is the fundamental characteristic that differentiates a worm from a virus, since the latter requires infecting a host program in order to replicate.

ing UDP). Most exploit-based worms perform target acquisition by randomly scanning the IP address space and probing the hosts at the generated addresses for the intended vulnerability. To optimize the propagation speed, worms have employed a topological scanning technique which probabilistically favors generating local addresses (e.g., addresses in the same Class B subnet as the current host), under the assumption that nearby hosts are more likely to have the same vulnerability. As another optimization, worm propagation can begin with a hit-list of known targets, which are used to bootstrap the spread of the worm [181].

Email-Based Worm. Email-based worms spread by attaching the worm binary to an email message and convincing the recipient to execute the program. Worms in this class often make use of social engineering techniques [132] to dupe unwitting users into executing the worm. A common method is to spoof the address in the email `From:` field header to make the message appear to be sent by an official or a contact of the intended recipient. Another technique is to hide the worm executable extension (`.exe`) by appending it to an alternate extension which is associated with a different file type (e.g., `.jpg`).[2] Furthermore, the most effective worms craft email messages with specific and enticing content that is designed to pique the victim's interest. Email-based worms may perform target acquisition by generating random email addresses, but they typically mine the list of email contacts on the host and send email messages with the worm attachment to those entities.

[2]This technique is made more effective due to a feature in the Windows operating system that hides the file extension by default when files are viewed (e.g., `worm.jpg.exe` is displayed as `worm.jpg`).

Web-Based Malware

Exploit-based worms demonstrate that vulnerabilities in popular network-based services are commonly exploited by attackers. Similarly, the web browser has become a ripe target for exploit by web-based malware [151]. While services are now better protected from attack due to operating system security enhancements,[3] the browser remains a common point of attack, for three reasons. First, many users do not update their web browser in a timely fashion, which allows vulnerabilities to persist across a large population. For example, many enterprise deployments require the use of older (and vulnerable) browser versions to maintain compatibility with outdated line-of-business web applications. Furthermore, home users often simply ignore or forget to apply browser updates. Second, hosts behind network address translation (NAT) devices or web proxies are easier to attack through a web browser. This is because victims themselves visit malicious websites, which force the download and execution of the exploit code in their browsers. Finally, web-based attacks are hard to detect because they blend in with normal web traffic.

Web-based malware typically takes the form of maliciously crafted client-side script (e.g., JavaScript) that is executed in the context of the web browser process. The script attempts to exploit a vulnerability in the browser or one of its plug-ins that results in arbitrary code execution. This code usually downloads and executes additional malware components to the host, which carry out the goals of the attacker [146]. Since the attack begins with the user simply navigating the web browser to a malicious site, it is

[3] The most important addition is the improved Windows Firewall service, which comes enabled by default in Windows XP Service Pack 2.

known as a *drive-by download* attack [150].

Trojan Horse

A Trojan horse is a malicious program that is disguised to appear to be legitimate, typically by including (or at least advertising) useful or otherwise interesting functionality. Trojan horse programs were first distributed in the early 1980s on Bulletin Board Systems (BBSs) [32]. Users of BBSs were persuaded to download and execute the malicious programs under the pretense that each was a new game or utility. Modern Trojan horses are usually sent as attachments to email messages. Thus, like email-based worms, Trojan horses use social engineering to abet their execution. In fact, email-based worms can be thought of as Trojan horses that also include a propagation component.[4]

Other Trojan horses are designed to hide malicious functionality in popular applications. For example, in Ken Thompson's Turing Award lecture, he describes a proof of concept Trojan horse that adds a backdoor to the traditional Unix `login` command [193]. The backdoor allows an attacker to login to the system as any user by authenticating with a special password. The additional malicious functionality is made possible by another Trojan horse that is added to the compiler. The second Trojan identifies when a benign version of the `login` command is being compiled and inserts the backdoor into it. This makes it very difficult to detect the malicious code additions, as the `login` program source code does not contain evidence of the backdoor.

[4]The subtle distinction is that worms necessarily self-propagate while Trojan horses might not (e.g., the early BBS Trojans do not).

1.1.2 Malware as a Concealment Mechanism

After malware is successfully delivered to the target host, its next step is often to hide its presence. Concealment is important because the payload components which are installed by malware represent income opportunities that attackers wish to protect from being detected (malware payloads are discussed in Section 1.1.3). To this end, malware authors have made significant advances in a type of malware known as a *rootkit*.

A rootkit is a set of tools that an attacker installs on a compromised host in order to conceal all evidence of the attack itself, as well as the other malware components that it brings onto the system. For example, rootkits may include a program that edits system logs to remove indications of the intrusion. Most often though, rootkits consist of tools that are designed to allow the attacker to retain control of a compromised system in a covert manner. Specifically, rootkits install software components that hide malicious processes from diagnostic utilities, such as `ps` or `top` on Unix-like operating systems or `Taskmgr.exe` on the Windows OS. Rootkits that exhibit this functionality are divided into three classes: *user-mode*, *kernel-mode*, and *virtual machine-based*.

User-Mode Rootkits. User-mode rootkits are malicious components that execute entirely at the user-mode (Ring 3) processor privilege level as stand-alone programs or as components within existing applications. The simplest user-mode rootkits are composed of a set of Trojan horse versions of standard system utilities. These malicious tools are installed as replacements for their benign counterparts, and they are designed so that when they are invoked, they hide the presence of malicious files and processes.

For example, a malicious version of `ls` and `ps` might be installed in the Unix OS so that malware will be excluded from directory and process listings, respectively.

Another type of user-mode rootkit makes use of hooking techniques, which allow the rootkit to interpose on function calls and modify their return values. Take, for example, Import Address Table (IAT) hooking in the Windows operating system. In Windows, each executable and dynamic-link library (DLL) has an IAT that is used to resolve function addresses when an application makes calls to imported Win32 APIs (e.g., APIs exported by `Kernel32.dll`). A rootkit can overwrite an entry within the IAT so that a function of the attacker's choosing is invoked instead of the requested API. For example, a rootkit may hook the entry for the `FindNextFile` API so that whenever the application attempts a directory listing, the rootkit can filter malicious files from the listing result.

Kernel-Mode Rootkits. It is relatively easy to detect user-mode rootkits by running tools that perform file system integrity checking [90], application signature matching [137], or system information differencing [24]. Furthermore, user-mode hooking is cumbersome because the rootkit must hook functions in each targeted process individually, which may include monitoring the system to hook new processes as they begin executing. Kernel-mode rootkits are more robust because they reside in the kernel (Ring 0) and, therefore, have direct access to kernel data structures, which can be modified to conceal malicious processes system-wide.

There are three strategies that kernel-mode rootkits employ to hide other malware. First, kernel-mode hooking, similar to its user-mode counterpart, is a technique that is

used to hook kernel data structures to subvert the normal operation of function calls. The canonical example is System Service Dispatch Table (SSDT) hooking. The SSDT is a Windows kernel structure which maintains a lookup table that the kernel uses to locate native system services (system calls). A rootkit can modify the service addresses in the table to point to proxy functions that it provides. Subsequently, whenever a hooked system service is requested, the rootkit function is invoked instead, which typically forwards the call to the original routine and then filters the result before returning it back to the caller.

Second, rootkits install filter drivers that layer on top of system device drivers. Ironically, this is the same technique that many antivirus tools use to scan for malicious files. A rootkit can, for example, install a driver that interposes on the input and output that is directed to and from the file system driver. Then the rootkit filter driver can modify the requests to and responses from the file system driver.

The third kernel-mode rootkit technique is called Direct Kernel Object Manipulation (DKOM). This strategy focuses on altering objects that the kernel uses for system accounting and auditing. For example, the Windows kernel maintains a list of objects that represents currently running processes (`PsLoadedModuleList`). A rootkit can modify this list so as to unlink a malicious process entry. Consequently, when applications make queries to functions that use the process object list (e.g., `NtQuerySystemInformation`), the unlinked process is effectively hidden.

Virtual Machine-Based Rootkits. Kernel-mode rootkits are difficult to detect because they control the interfaces that describe the state of the operating system. That

said, rootkits do leave artifacts behind when modifying kernel objects and data structures, which can be used to betray their existence [164]. Virtual machine-based rootkits have been proposed as a technique to allow rootkits to fully control a compromised operating system while remaining completely undetected [94, 165]. Rootkits accomplish this by installing a virtual machine monitor (VMM) and hoisting the operating system into a virtual machine that is then controlled by the VMM. Thus, the system becomes a guest OS that is trapped in a virtual machine and unaware of the rootkit controller, while the rootkit can utilize the original host operating system to run malicious applications and services.

1.1.3 Malware Payloads

The previous sections discussed how malware is delivered to a target host and the techniques that are used to conceal the presence of malware once it is installed. After gaining access to the compromised system, malware focuses on fulfilling the attacker's intent. Historically, attackers only sought notoriety, so their malicious software was programmed simply to propagate as fast as possible without any other direct impact [133] or to carry annoying payloads that, for example, deface websites [134] or delete files from the hard disk [172]. Malware technology has evolved to concentrate on leveraging a compromised host (and the data it contains) for profit. This section describes malicious components that attackers deploy to realize this goal.

Spyware

Spyware is a type of malware that runs in a covert manner in order to monitor a user's actions and collect sensitive information without the user's knowledge. The spyware then exports the gathered information to an external entity. *Adware* is a form of spyware that observes a user's interests and behavior (e.g., by monitoring the user's web browsing habits) in order to send the user targeted advertisements. Other kinds of spyware are more malicious in nature and extract a user's personal information and banking credentials in order to sell them in the underground market [50]. Spyware is typically implemented in one of two ways: as an extension to the web browser or as a *keylogger*.

Browser Extension Spyware. All modern browsers support extensions that communicate with the browser through a documented API. For example, Internet Explorer allows plug-in modules known as Browser Helper Objects (BHOs) to extend the functionality of the browser. A BHO makes use of interfaces through which it can request access to the contents of the current page and its URL as well as register for various events that are related to the user's browsing session. Thus, spyware BHOs are able to access any sensitive data that the user enters into web pages while navigating with the browser.

Keylogger. Spyware also utilizes keylogging techniques, which track a user's individual key presses to obtain the data that a user enters at the keyboard. Keyloggers employ several strategies in order to monitor a user's data. First, a keylogger can

be implemented as a *packet sniffer*, which promiscuously intercepts and logs network traffic. Spyware might use a packet sniffer to retrieve HTTP POST data or other unencrypted information that has been sent on the wire. Second, a keylogger may be a *form grabber*, that is, a software component that executes in the context of the web browser process (e.g., as a BHO) and captures HTML form data that is entered by a user. The third technique adopted by keyloggers is to use APIs provided by the operating system. For example, on the Windows platform, a keylogger can register for keyboard events by making use of Windows hooks (e.g., `WM_KEYDOWN`). This allows the keylogger to transparently intercept and log all key presses. Finally, a keylogger may be implemented as a kernel extension that filters the keyboard hardware interrupts that are handled by the keyboard device driver.

Direct Payment Malware

Attackers frequently leverage spyware to profit from their victims, but monetizing the gleaned personal information requires miscreants to do business in an unpredictable underground marketplace [48]. Therefore, malware authors have also found ways to earn direct payment from their attacks. This section describes two popular types of direct payment malware: *ransomware* and *scareware*.

Ransomware. Ransomware is malware that effectively holds a compromised machine, and, in particular, the data that it hosts, for ransom. This form of malware was originally called a *cryptovirus* [214] since it uses cryptographic primitives to carry out the attack. After gaining access to a host, ransomware encrypts a selection of files on

the hard drive with a key that is unknown to the victim. Typically, the attacker leaves behind a text file that provides instructions that detail how and where to pay the ransom in exchange for the secret key to decrypt the files.

Scareware. Scareware is a type of malware that uses social engineering techniques to convince victims of an impending (but actually fictitious) threat. A common form of scareware is rogue security software, which impersonates a legitimate security tool (e.g., an antivirus scanner), but does not provide genuine protection [28, 184].[5] Nonetheless, the rogue security software usually purports to have detected malicious threats on the victim's computer and will "remove" them upon payment.

Bot Malware

A malicious bot is a program that is designed to infiltrate a compromised host and report back to a control center, which is operated by an attacker, to await further commands. A network of bots is formed when they each report to the same command and control server. These networks are called *botnets*. Botnets represent the state-of-the-art in malware technology because of their potential to do harm, due to their scale (botnets may contain millions of bots [147]) and flexibility (bots support a diverse set of commands [22]).

Botnet Functionality. A malicious bot may be programmed to support a wide range of functionality, which is available to the botnet controller on-demand. For example, a

[5] In fact, many instances of rogue security software actually facilitate the installation of malware.

bot can act as any of the malware payloads that were described in the previous sections (spyware, ransomware, and scareware). Furthermore, bot commands can be parameterized, which allows for even more advanced (and profitable) attacks.

One such function is *click fraud*, an attack which exploits the pay-per-click advertising model on the web. Click fraud occurs when an automated program is deployed to impersonate a legitimate user of a web browser in order to "click" on an online advertisement (or ad) for the sole purpose of generating a payment associated with that click. Attackers utilize bots to orchestrate click fraud in one of two ways [33]. First, bots may be instructed to click on ads that appear on web pages that are controlled by the attacker. In this way, the attacker earns a percentage of the money which is paid by the advertiser for each click. This attack is known as *publisher click fraud*. In the second attack, called *advertiser click fraud*, bots are commanded to click on the ads of a competitor. After a threshold of clicks, the competing advertiser's budget will be exhausted and the advertiser will no longer display ads. At this point, the attacker's ads are more likely to be shown and, thus, business is driven to the attacker's site.

A second advanced attack that is made possible by botnets is distributed *denial-of-service* (DDoS). A DDoS attack is a distributed effort to remotely deny access to an Internet resource (e.g., a web server), by making it unavailable to its intended users. Since botnets have the capacity to consume large amounts of bandwidth, they are frequently deployed to carry out DDoS attacks [52]. To accomplish the attack, a botnet is configured to direct many illegitimate network connections to the target service, which overwhelms the resource that hosts the service and renders it unusable. DDoS attacks are carried out for three primary reasons. First, attacks are leveraged against popular

websites as a form of protest, often due to political opposition. Second, DDoS is used by unscrupulous businesses to attack their direct competitors. Finally, attackers use the threat of DDoS to extort payments from businesses with a strong online presence. For example, offshore gaming sites are often targeted immediately preceding popular sporting events [123]. Commonly, the businesses pay the extortion fee because the cost is low when compared to the amount of revenue that would be lost should their site be made inaccessible.

Botnets are also commonly leveraged for sending spam emails as part of large-scale spam campaigns [102, 145]. The botnet controller sends a command that includes a template from which to build the spam messages and a target list of email recipients (or instructions for how to obtain such a list). Bots, in turn, use the template to construct the actual spam messages and deliver them to the recipients. Spam campaigns have a variety of goals, including phishing scams, pharmaceutical sales, stock scams, and malware distribution, to name a few.

Botnet Command and Control. As mentioned previously, botnet controllers instruct bots to carry out a variety of sophisticated attacks through a communication channel for command and control (C&C). As botnet technology has evolved, so too have the C&C channels, in order to afford the attacker flexibility, reliability, and resistance to detection. C&C channels can be divided into two categories: *centralized* and *decentralized*.

Early botnets utilized centralized C&C channels over the Internet Relay Chat (IRC) protocol. IRC is a real-time messaging protocol that coordinates group communication

over the Internet. IRC has a large user base, and there are many freely available public servers where users self-coordinate into discussion forums called *channels*. Malware authors use IRC for C&C for two reasons. First, many open source benign bots have been developed for IRC [38]. These bots provide users with useful services, such as managing nickname registration (NickServ) or IRC channel maintenance (ChanServ). This makes it easy for bot authors to modify the code base of an existing bot to add malicious functionality. Second, IRC is a relatively simple protocol, and it is straightforward to embed malicious commands within it. To make use of IRC-based C&C, malicious bots are directed to join a particular IRC channel and monitor the channel events. The botnet controller then issues a command to all or a subset of the bots by broadcasting a group message, changing the channel topic, or sending each bot a private message.

Attackers also commonly use the Hypertext Transfer Protocol (HTTP) for botnet command and control. Like IRC, the HTTP protocol operates within a centralized, client-server communication model. Since HTTP is the fundamental protocol for communication on the World Wide Web, bots can be written as simple web clients that contact a web server for C&C. Bots periodically make HTTP requests to a web page on a server that is operated by the botnet controller. The server replies with a command that is embedded in the HTTP response.

Some botnets make use of a decentralized peer-to-peer (P2P) protocol for C&C. A P2P network follows a different model than a traditional client-server architecture. In the latter, a server is the supplier of content and services, while the clients are the consumers. By contrast, in a P2P arrangement, all network participants act as both

client and server. A botnet controller can leverage the P2P model for botnet C&C by providing a command to a small number of bots and allowing them to self-organize in order to propagate the command to the rest of the botnet.

Botnet C&C Discussion. There are advantages and disadvantages associated with using IRC, HTTP, and P2P for botnet C&C. IRC-based C&C can be characterized as push-based, since bots wait in an IRC channel and respond to real-time commands. This is an advantage over HTTP-based C&C, which is pull-based, because when using HTTP, bots must periodically poll a web server to receive a command update. Polling introduces communication overhead when the command has not changed, and it adds delay because a bot will not be able to immediately recognize when the botnet controller has issued a new command. P2P-based C&C protocols can be push-based or pull-based, depending on how the bot receives command notification.

In general, botnets utilizing centralized C&C protocols (IRC and HTTP) are easier to take down than those using a decentralized approach (P2P). This is because once a centralized server is located, it can be disabled by the hosting provider [101]. Furthermore, in the case of IRC, the server administrator can simply close the IRC channel that is used for C&C. Alternatively, the DNS resolver which provides the mapping for the IP address associated with the C&C server can be updated to sinkhole bots to a benign server or nonexistent address [30]. Decentralized approaches do not suffer from these problems. Disabling one bot (acting as a server) only removes a single node from the P2P network, and the rest of the network can reorganize to continue relaying commands.

Finally, HTTP-based C&C is more difficult to detect and block than IRC-based and P2P-based approaches for two reasons. First, as stated before, the web is nearly ubiquitous and HTTP is its fundamental protocol. In contrast, IRC and P2P protocols are not as widely utilized, so their use may arouse suspicion. Second, it is difficult to differentiate HTTP-based bot traffic from legitimate HTTP connections. Many firewalls are configured to block network traffic to and from unconventional ports, including the ports which are commonly used by IRC and P2P protocols [192]. HTTP-based bots subvert these firewall rules because they use the most popular protocol on the Internet, which cannot be reasonably blocked.

1.2 Stages of Malware Defense

A variety of defense techniques have been proposed to combat the ever-evolving malware threat. These approaches can be generally divided into three stages: *analysis*, *detection*, and *response*. Each stage informs the next, and combining techniques from each is a strong basis for building effective security tools. This section presents the stages of malware defense and discusses benefits and drawbacks of the various techniques within each category.

1.2.1 Malware Analysis

Malware analysis is the methodology that is used to discover the intent, functionality, structure, or behavior of malicious software. Malware analysis is a crucial component

of malware defense, because it informs detection strategies and provides the foundation for the development of effective tools to combat malware. Techniques for malware analysis can be classified into two broad categories: *static* and *dynamic*. Static analysis focuses on examining the malware code without executing it. In contrast, dynamic analysis determines malware actions and characteristics during run-time. While they are presented separately, the two forms of malware analysis are complementary and the techniques applied to one inform and improve the effectiveness of the other.

Static Analysis

The goal of static malware analysis is to discover the actions that malware will take during run-time by evaluating the binary code offline instead of executing it. While manual static analysis can be performed by using a tool such as a binary disassembler or decompiler, this is a time consuming and error prone process, so automatic techniques are typically used. Static analysis of malware can be viewed as an application of the techniques that have been developed for static program analysis. To this end, static malware analysis has made use of standard techniques such as model checking [92], symbolic execution [104], and program slicing [100] in order to learn more about the behavior of a malware sample.

The advantage of static analysis techniques is that they have the potential to discover all code paths of a malware program, and, thus, all of the behavior of a malware instance can be captured. This is an important benefit because many malware instances exhibit behaviors that trigger only when a certain condition holds, such as performing an action

on a specific date or responding to a particular bot command. Unfortunately, malware authors are aware of the power of static analysis, and they utilize advanced obfuscation techniques [111], binary packing [66], polymorphic encryption [182], self-modifying code, and instruction set emulation [174] to make analysis more difficult. Thus, the main disadvantage of static analysis is that the code available at compile-time is not necessarily the same code that will execute at run-time.

Dynamic Analysis

Dynamic analysis techniques execute malware and observe its actions and the effects that it has on the system. Typically, the analysis is done in an instrumented testing environment that provides a means to systematically monitor malware behavior. Furthermore, in the case of malware with a "phone home" capability, such as spyware and malicious bots, the test harness may include the impersonation of services that the malware depends upon (e.g., an IRC server for a bot that communicates over IRC-based C&C). In any case, dynamic malware analysis can be done manually or in an automated fashion.

Manual dynamic analysis typically entails deploying a virtualized environment and executing a malware sample under a debugger. In this way, the analyst can step through the malware code in order to determine its behavior. Additionally, general system monitoring tools can be used to observe, for example, the files that the malware instance puts on the file system, the registry keys that it sets, or the network connections that it makes. A complementary method of dynamic malware analysis takes a server-centric

approach instead of focusing on malware as it runs on the client host. That is, after an analyst determines where an instance of spyware or a malicious bot will connect, a server can be deployed to impersonate the remote host. This may provide further insights into the intent of the malware.

As in the case of static analysis, manual dynamic analysis of malware is a tedious process that takes considerable time and effort. Advances have been made in automatic approaches in order to quickly provide an accurate summary of the behavior of many malware instances. For example, monitoring environments have been developed to provide malware with the illusion that it is executing on an actual victim host, while, in fact, its behavior (e.g., the processes that it spawns or the network connections that it makes) is being observed in a sandbox [138, 10, 208]. At the end of the analysis session, a report is generated, and an analyst can gather a summary of the behavior of the malware, which may help guide further static analysis. In fact, using dynamic analysis to enable better static analysis of malware is an important goal in and of itself. For instance, dynamic analysis tools have been devised that use heuristics to determine when an executing malware instance has been unpacked [162, 83, 119, 175]. Since malware obfuscation is the primary impediment to static analysis techniques, finding this execution point is beneficial, as the deobfuscated malware image can be dumped and further analyzed statically.

The main advantage of dynamic analysis techniques is that they observe the actual actions of the malware in execution, and, thus, represent a ground truth of malicious behavior. This is in contrast to static malware analysis, which can only infer behavior and, thus, can be made difficult or impossible through dynamic code rewriting or

obfuscation techniques. However, the result of a dynamic analysis session is incomplete because it only represents a single path of execution, which likely fails to catch important, trigger-based malware behavior (e.g., time-sensitive actions). Furthermore, malware has adapted to detect when it is executing in an emulated or virtualized environment [54] or is being run under a debugger. As a result, the malware either terminates prematurely or refuses to exhibit malicious behavior while under analysis. The research community is aware of these problems, and dynamic analysis techniques have been proposed to mitigate each: for example, [135] for the former and [5] for the latter.

1.2.2 Malware Detection

Malware detection is the process of monitoring events on a host or in the network in order to identify malicious programs. Malware detection tools depend upon analysis techniques to build detection models that are applied to a set of events in order to determine if the events indicate the presence of malware. The detection models are constructed on the basis of signatures or behaviors that are representative of malware. Detection models are applied to events in one of two monitoring domains: in the network or on the host. These variants are discussed next.

Network-Based Detection

Network-based approaches to malware detection operate in the network and observe events as streams of network data. In this domain, often times detection models are built from signatures, which are matched against known sequences of bytes in network

packet contents that are representative of malware. For example, some network-based *intrusion detection systems* make use of signatures that are crafted manually [160, 144] or automatically [91, 178], and malicious bot detectors [64, 211] make use of this strategy as well. Additionally, malware can be detected in the network using behavior-based approaches. Typically, this entails learning what constitutes normal network behavior and detecting deviations from this model as malicious. For example, anomaly-based intrusion detection systems have applied statistical models [202] and bot traffic clustering [63] to detect malware.

The advantage of deploying a detection tool in the network is that a single sensor can observe multiple endpoints. Unfortunately, this convenience comes at a cost. First, resource-intensive detection tools may have trouble in high-bandwidth deployments. Second, detecting malware in the network is indirect. That is, a detection system can only infer the presence of malware by monitoring its traffic. This is problematic because malware can encrypt its traffic to evade signature-based models or can use covert channels or blending techniques [45] to make its traffic appear normal. Finally, malware does not have to use the network at all, which would render a network-based detection tool useless.

Host-Based Detection

Host-based malware detection tools are deployed on an individual endpoint. Traditionally, malware detection on the host is performed by statically scanning executable files to match sequences of bytes or instructions to sequences that are indicative of a known

malware instance [105]. The use of syntactic signature detection models are the de facto standard in the antivirus industry, but they have become ineffective in the face of malicious programs that utilize advanced obfuscation techniques [111], polymorphic encryption [182], and instruction set emulation [174] (recall Table 1.2). Behavior-based detection models have been proposed that construct semantic signatures [23], which act as a template for malicious behavior. Unfortunately, due to their reliance on static analysis techniques, these approaches are vulnerable to evasion by binary packing [66] and self-modifying code. Techniques have been developed to automatically unpack malicious binaries [162, 83, 119, 175], but they are not perfect, and advanced obfuscation techniques remain a problem [136].

To overcome the obstacles that exist when using static techniques, dynamic approaches to host-based malware detection have been proposed. For example, some systems make use of dynamic data-flow (taint) analysis to capture malware behavior, which can be used for detection [96, 37, 213]. Furthermore, host-based anomaly detection systems monitor sequences of system calls executions [47, 199, 103] and create an alert when patterns of calls are detected that deviate from the model of normal behavior. The problem with this approach is that it is easily evaded by reordering independent system calls or adding spurious calls to the execution trace.

Host-based malware detection has some advantages. First, these detection schemes are able to observe all actions of a malware instance, which make them easier to monitor. Second, static approaches have the capability to detect malware even before it executes (the shortcomings of static techniques withstanding). However, there is a higher cost of deployment of host-based tools, which can make maintenance an issue.

1.2.3 Malware Response

Malware response is the process of reacting to a detected malware instance. The response could be as simple as writing an entry in a system log or displaying a warning dialog box or as sophisticated as automatically deploying a firewall rule that will disallow similar malware samples from reaching the host or network. Of course, if malware is not detected, there can be no response, so malware response is tightly coupled with detection techniques. Strategies for malware response can be divided into a three groups: *notification*, *containment*, and *prevention*.

Malware Notification. Malware notification is simply making a user or administrator of the system aware that the detection tool has (possibly) identified a malicious program. This is the mechanism that is employed by most commercial antivirus tools in a conservative configuration; a file scan detects a malware instance and a dialog box is presented to the user. The dialog box shows details of the scan result and suggestions for further action, such as file quarantine or removal. In the case of large-scale intrusion detection systems that generate many alerts, this dialog box notification strategy is not reasonable. Therefore, such systems write an entry in a system log each time an alert is created. Alert correlation techniques [196] can then be applied to the logs of one or more deployed intrusion detection systems to provide a succinct and high-level view of detection events.

Malware Containment. Malware containment techniques represent a broad category of approaches for disabling or otherwise hindering a detected malware instance.

In a host-based setting, this typically means terminating the process that is hosting the malicious executable (if the malware is already running at the time of detection) and moving it to a quarantined location on disk or deleting it from the file system. As mentioned above, this action may occur as a user-initiated response to an alert dialog box, or it may happen automatically if the detection tool is more aggressively configured [125]. Furthermore, containment refers to techniques that are designed to disrupt a malicious program's workflow by limiting access to the resources that it requires to carry out its intent (e.g., the file system or network). A broader view of malware containment entails techniques that limit the effectiveness of malware attacks on the Internet. For example, a method has been proposed to limit the spread of exploit-based scanning worms that make too many network connections [207]. Furthermore, strategies have been devised to disrupt the communication channel of a peer-to-peer botnet by poisoning its command and control infrastructure [71] or impersonating many (fake) bot clients [34].

Malware Prevention. Malware prevention techniques aim to preemptively block a malware instance from infecting a host. For example, network-based intrusion detection systems that are deployed in-line (as opposed to using a network tap or port span, for example) are often augmented with the capability to log an alert and drop the traffic that is associated with a detected malware intrusion. Intrusion detection systems that are extended with such a preventative component are sometimes called *intrusion prevention systems* [169].

Malware response is an important step in a complete malware defense solution, but

there are some potential drawbacks. First, false positives can be detrimental and possibly dangerous to malware response. Certainly, an incomplete detection technique is to blame for the false positive in the first place, but the effect can be amplified by a faulty response. For example, consider an antivirus tool that mistakenly classifies a critical system file as a malware instance and automatically removes it from the host [190, 31]. Second, containment and prevention responses can be exploited by attackers to induce a denial of service. For example, a network-based service can be attacked by malware in order to induce a response that would terminate the service.

1.3 Contributions to Malware Defense

While much work has been done in the areas of malware analysis, detection, and response (as alluded to in the previous section and reviewed further in Chapter 2), there are still opportunities for further research. Next we discuss some of the open problems and briefly outline our approaches to solving them.

1.3.1 Analysis: A Botnet Takeover

Dynamically analyzing a malware sample in a sandboxed environment [138, 10, 208] can be a good first-order approximation of the sample's behavior. Unfortunately, this sort of analysis can only report on the results that are obtained from a single execution of a malware instance. Of course, the sample can be executed repeatedly in the hopes of finding additional behavior, but this is a cumbersome approach, and it is unlikely to

discover trigger-based behaviors, such as time-sensitive actions (e.g., a time bomb or receipt of remote commands).

In addition, analyzing a malware sample in a controlled environment can only offer a limited amount of information with respect to its purpose or impact. The state-of-the-art of advanced threats today include large-scale botnets that have a wide breadth of malicious goals, such as denial of service, theft of personal information, and delivery of spam emails (as discussed in Section 1.1.3). Without analyzing these malicious networks in the wild, it is impossible to gain insights into how large of a problem they pose, and, by extension, how resources should be deployed to defend against them.

To this end, we coordinated the takeover of the sophisticated Torpig botnet for a period of ten days. In so doing, we were able to analyze the number of infected victims to obtain an accurate estimation of the botnet's active population. Additionally, we were given a unique opportunity to gain insights into the types and volume of personal data that a large-scale botnet affords to attackers.

1.3.2 Detection: Dynamic Code Identity Tracking

Malware technology has evolved significantly in order to avoid detection, for example, by utilizing sophisticated code obfuscation techniques. Furthermore, advanced malware is able to tamper with legitimate programs and inject malicious functionality into otherwise benign processes (e.g., through a drive-by download or a remote DLL injection). Since static blacklisting and whitelisting approaches fail to reliably detect malware in either of these cases, more advanced techniques are required.

Code identity is a primitive that allows an entity to recognize a known, trusted application as it executes [143]. This primitive supports trusted computing mechanisms such as sealed storage [39] and remote attestation [166]. Unfortunately, there is a generally acknowledged limitation in the implementation of current code identity mechanisms in that they are fundamentally static. That is, code identity is captured at program load-time and, thus, does not reflect the dynamic nature of executing code as it changes over the course of its run-time. As a result, when a running process is altered, for example, because of an exploit or through injected, malicious code, its identity is not updated to reflect this change.

To solve the code identity problem, we have developed DYMO, a system that provides a *dynamic* code identity primitive that tracks the run-time integrity of a process and can be used to detect code integrity attacks. To this end, a host-based component computes an *identity label* that reflects the executable memory regions of running applications (including dynamically generated code). These labels can be used by the operating system to enforce application-based access control policies. Moreover, to demonstrate a practical application of our approach, we implemented an extension to DYMO that labels network packets with information about the process that originated the traffic. Such provenance information is useful for distinguishing between legitimate and malicious activity at the network level.

1.3.3 Response: Process Interaction Tracking

Modern operating systems provide a number of different mechanisms that allow processes to interact. These interactions can generally be divided into two classes: interprocess communication techniques, which a process supports to provide services to its clients, and injection methods, which allow a process to inject code or data directly into another process' address space. Operating systems support these mechanisms to enable better performance and to provide simple and elegant software development APIs that promote cooperation between processes.

Unfortunately, these process interaction channels introduce problems at the end host concerning malware *containment* and *attribution* of malicious actions. In particular, host-based security systems rely on process isolation to detect and contain malware. However, interaction mechanisms allow malware to manipulate a trusted process to carry out malicious actions on its behalf. In this case, existing security products will typically either ignore the actions or mistakenly attribute them to the trusted process. For example, a host-based security tool might be configured to deny untrusted processes from accessing the network, but malware could circumvent this policy by abusing a (trusted) web browser to get access to the Internet. In short, an effective host-based security solution must monitor interactions between processes, and current systems are lacking in this regard.

To address this problem, we have developed PRISON, a system that tracks process interactions and prevents malware from leveraging benign programs to fulfill its malicious intent. To this end, a kernel extension intercepts the various system calls that enable

processes to interact, and the system analyzes the calls to determine whether or not the interaction should be allowed. PRISON can be deployed as an online system for tracking and containing malicious process interactions to effectively mitigate the threat of malware. The system can also be used as a dynamic analysis tool to aid an analyst in understanding a malware sample's effect on its environment.

1.3.4 Summary of Contributions

To summarize, in this dissertation we make the following contributions to the area of malware defense:

- We discuss our analysis of the operation of the Torpig botnet. In particular, we emphasize our analysis of the data we obtained that was sent by more than 180,000 infected machines, as it provides insight into the nature of the threat that botnets in general, and Torpig in particular, present to the Internet.

- We present the design and implementation of DYMO, a system that tracks the dynamic code identity of the executables that are running on a host. DYMO detects and mitigates stand-alone malware as well as malicious code that tampers with legitimate programs.

- We describe the design and implementation of PRISON, a system that contains malware by preventing a malicious process from interacting with another process to carry out actions on its behalf. PRISON can mitigate attacks that evade other host-based malware defense systems. Furthermore, our system can by used as

a dynamic malware analysis tool to aid in understanding malicious interaction behavior.

Our work makes substantial contributions to the area of malware defense, addressing all three stages that we described in Section 1.2. In particular, our takeover of the Torpig botnet illustrates an example of dynamic *analysis* of active malware in the wild. Furthermore, DYMO represents a host-based system that implements a dynamic code identity primitive, which provides malware *detection* capabilities to sensors on the host or in the network. Finally, PRISON is a host-based system that *contains* malicious processes by analyzing and blocking their malevolent interactions with trusted applications. Additionally, PRISON can be used as a dynamic malware *analysis* tool that captures a malware sample's effect on its environment. Taken together, these contributions advance the state of the art of malware defense and provide the basis for interesting future work in the area.

The remainder of this dissertation is organized as follows. Chapter 2 presents an overview of related work in the field of malware defense. Chapter 3 discusses our takeover of the Torpig botnet. Chapters 4 and 5 introduce the design of DYMO and PRISON, respectively. Finally, Chapter 6 concludes.

Chapter 2

Background and Related Work

The domain of malware defense is broad, and a considerable amount of research has been executed in the areas of malware analysis, detection, and response. Given this breadth, the scope of this chapter is limited to the work in malware defense that is most closely related to the contributions that we make in this dissertation. In particular, the following sections discuss research efforts that are pertinent to the observation of actively deployed malware, the monitoring of process interactions, and the tracking of process integrity.

2.1 Active Malware Observation

Observing the operation of malicious software during execution is an important form of dynamic malware analysis. It is undecidable to statically determine if a sample will

behave maliciously [108], so monitoring the behavior of malware at run-time represents a ground truth. Furthermore, it is difficult to grasp the impact of advanced bot malware by analyzing a single sample in isolation. In contrast, by examining botnets in the wild, an analyst can gain tremendous insight into the threat. This section introduces approaches that are related to active malware observation. The techniques are divided into two categories: instrumented environments for dynamic malware analysis and applications of live botnet analysis.

2.1.1 Instrumented Environments

Manual analysis of a malware instance is tedious and error-prone. As a result, a number of systems have been developed to automatically execute a sample in a closed environment in order to monitor its behavior (e.g., the files it creates, the registry keys it sets, or the network connections it opens). At the end of the analysis session, the system outputs a detailed report that describes the sample's interaction with the environment. This report can then guide classification or further manual analyses of the sample.

One such system is the Norman Sandbox [138]. Norman emulates a Windows host by implementing core components, such as the BIOS and ROM, and simulates its surrounding operating environment, including the network. Another system, CWSandbox [208], operates as a user-mode process on a true Windows host and monitors system call requests by launching the malware instance in a suspended state and in-line hooking its interesting API calls (e.g., `CreateFile` and `VirtualAlloc`) before

the sample begins execution. Like CWSandbox, TTAnalyze [1] [10] also hooks system calls, but it runs a fully emulated operating system under the QEMU emulator [12]. The emulator offers TTAnalyze the flexibility to observe the entire system state without modifying the host. Finally, Wepawet [27] is an approach to the detection and analysis of web-based malware. Wepawet uses machine learning techniques to establish a baseline of normal JavaScript execution behavior and then monitors a given sample to determine if it deviates from that model.

2.1.2 Live Botnet Analysis

The previously described approaches offer an effective way to gain insights into the behavior of modern malware, but they can only determine the characteristics of a single sample in isolation. To bridge this gap, the authors of TTAnalyze mined their large malware sample dataset in order to get a high-level view of how malware generally behaves [9] and to effectively cluster malware behaviors [8]. An orthogonal approach is to infiltrate active botnets in the wild. From this vantage point, an analyst is able to deeply understand the motivation and purpose of sophisticated attacks.

One approach is to perform passive analysis of the secondary effects that are caused by the activity of compromised machines. For example, researchers have collected spam emails that were likely sent by bots [220]. Using the emails, they were able to make indirect observations about the sizes and activities of different spamming botnets. Similar measurements focused on the DNS queries [153, 154] or DNS blacklist queries [156]

[1] TTAnalyze is currently maintained as the Anubis project [75].

that were performed by bot-infected machines. Other researchers analyzed network traffic (NetFlow data) at the tier 1 ISP level for cues that are characteristic of certain botnets (such as scanning or long-lived IRC connections) [86]. While the analysis of secondary effects provides interesting insights into particular botnet-related behaviors, one can typically only monitor a small portion of the Internet. Moreover, the detection is limited to those botnets that actually exhibit the activity that is targeted by the analysis.

A more active approach to study botnets is via infiltration. That is, using an actual malware sample or a client that simulates a bot, researchers join a botnet to perform analysis from the inside. To achieve this, honeypots, honey clients, or spam traps are used to obtain a copy of a bot sample. The sample is then executed in a controlled environment, which makes it possible to observe the traffic that is exchanged between the bot and its command and control (C&C) server(s). In particular, one can record the commands that the bot receives and monitor its malicious activity. For some botnets that rely on a central IRC-based C&C server, joining a botnet can also reveal the IP addresses of other bot clients that are concurrently logged into the IRC channel [26, 52, 153]. While this technique worked well for some time, attackers have unfortunately adapted, and most current botnets use stripped-down IRC or HTTP servers as their centralized command and control channels. With such C&C infrastructures, it is no longer possible to make reliable statements about other bots by joining as a client.

Interestingly, due to the open, decentralized nature of peer-to-peer (P2P) protocols, it is possible to infiltrate P2P botnets, such as Storm [183], a large botnet with the primary purpose of delivering spam. In order to mitigate the effect of the botnet, re-

searchers were able to add noise to the communication channel by impersonating many bot clients (a so called Sybil attack) [34, 71]. Furthermore, Holz et al. infiltrated the botnet in order to estimate its size, and they investigated a weakness in the communication protocol that allowed unauthenticated clients to poison the botnet commands by publishing bogus keys in the overlay [71]. Kang et al. also used Storm as a case study for P2P-based botnet node enumeration [82]. Finally, Kanich et al. infiltrated Storm for the purpose of measuring the conversion rate of spam [84]. In order to do so, the authors inserted their own bot clients into the botnet. These clients were programmed to rewrite spam templates before forwarding them on to the rest of the bots in the network. The resulting spam messages contained links to sites that were under their control, so they could evaluate the effectiveness of their campaigns.

To overcome the limitations of passive measurements and infiltration – particularly in the case of centralized IRC and HTTP botnets – one can attempt to takeover the entire botnet, typically by taking control of the C&C channel. One way to achieve this is to directly seize the physical machines that host the C&C infrastructure [17, 195]. Of course, this is only an option for law enforcement agencies.

Alternatively, one can tamper with the domain name service (DNS), as bots typically resolve domain names to connect to their command and control infrastructure. Therefore, by collaborating with domain registrars (or other entities such as dynamic DNS providers), it is possible to change the mapping of a botnet domain to point to a machine controlled by the defender [30].

Finally, several recent botnets, including Torpig [35] and Conficker [147], make use of

a C&C location strategy we call *domain flux*. Using domain flux, each bot periodically (and independently) generates a list of domains that it contacts. The bot then proceeds to contact the domains one after another. The first host that sends a reply that identifies it as a valid C&C server is considered genuine, until the next period of domain generation is started. By reverse engineering the domain generation algorithm, it is possible to preregister domains that bots will contact at some future point, thus denying access to the botnet controller (botmaster) and redirecting bot requests to a server under one's own control. This provides a unique view on the entire infected host population and the information that is collected by the botmasters.

2.2 Process Interaction Monitoring

Due to the relatively open communication architecture afforded to processes in modern operating systems, they can interact in a number of ways. Monitoring the interactions between executing processes is a beneficial analysis technique, because it can provide insight into how and why two processes communicate. In particular, a malicious process that attempts to exploit a trusted, legitimate service in order to surreptitiously carry out its malevolent intent can be detected and thwarted by such analysis. This section examines relevant mechanisms to help mitigate malicious process interactions, beginning with a background discussion of access control.

2.2.1 Access Control

Access control is a fundamental operating system security feature that enables an authority to mediate access to the resources that are available on the system. Access control is vital to system security because it ensures that an entity (called a *subject*) can only access the resources (called *objects*) that are permitted under a given policy, which is enforced by a secure mediator (called a *reference monitor*). The policy can be represented by an Access Control Matrix [106] that enumerates the access privileges of each subject with respect to all objects in the system. A discretionary access control (DAC) policy is one in which object access rights are assigned by the object owner. In a mandatory access control (MAC) policy, access rights are determined by the system, based on one or more attributes of the subjects and objects. DAC was created to allow users to protect the objects they own from other users (e.g., a user may wish to allow only certain users to read a file). MAC was designed to enforce an organizational policy (e.g., users with Secret clearance may not read documents that are marked Top Secret [11]).

There are two basic approaches for implementing access control: models based on access control lists (ACLs) and those based on capabilities. In an ACL-based model, a subject's access to an object is approved by the reference monitor only if the object has an entry in its ACL that grants the requested access to the subject [167]. In this scheme, an ACL can be thought of as a column in the Access Control Matrix. By contrast, in a capability-based approach, a subject obtains an unforgeable reference to an object (a capability) that grants the desired access [109]. This proves to the reference monitor that the requested access is permitted. A capability can be considered a row in the

Access Control Matrix.

As alluded to above, ACLs and capabilities optimize the use of the Access Control Matrix to make access control decisions by storing columns with objects or rows with subjects, respectively. ACL-based systems can quickly determine the access rights that a given object grants to subjects, and capability-based approaches efficiently surmise all of the access rights that a given subject has to a set of objects. Therefore, the choice to implement access control with one scheme over the other can be reduced to the practical consideration of which of the preceding is the most commonly requested operation [87]. Because workflows more frequently require the reference monitor to check which subjects have what access rights with respect to a given object, ACL-based models perform better and have seen the widest deployment. For example, Windows NT, Unix, and Mac OS all use ACL-based discretionary access control.

ACL-based access control is not a panacea, however, and there are some problems with the model as implemented in modern operating systems, which has prompted healthy debate [130]. The most pertinent issue concerning ACLs in operating systems such as Windows and Unix is *ambient authority*. Ambient authority is an access control feature by which a subject is given the (explicit) authority it requires to perform its tasks as well as any additional (implicit) authority that happens to be available. This can be problematic because in modern systems, a subject is approximated by its associated authenticated user (so called *user-based authorization*). Users typically have a wide variety of access rights, so processes (subjects) are implicitly given more permissions than they require to complete their task (a violation of the *principle of least privilege* [168]). In particular, malware that executes under a particular user account gains

access to all of the objects that the user may access. This can lead to sensitive data exfiltration and the ability of malware to utilize a benign process to perform actions on its behalf (an instance of the *confused deputy problem* [67]). Ambient authority has motivated research into a variety of capability-based systems, such as PSOS [140], EROS [173], and Capsicum [205], which avoid these issues. Furthermore, numerous approaches to process isolation have been proposed that can mitigate this problem; these are discussed next.

2.2.2 Process Isolation

Process isolation is an important mechanism that operating systems use to protect the code or data of a process from the faulty or malicious behavior of other processes. Contemporary operating systems implement process isolation by leveraging two mechanisms that are provided by the memory management hardware of the CPU. First, the abstraction of virtual memory maintains address space boundaries that enable processes to share physical memory without interfering with each other. Second, protection domains (rings) protect privileged resources from unauthorized tampering.

Unfortunately, a single process can rarely perform any useful function in and of itself. Therefore, inter-process communication mechanisms are supported to allow processes to exchange data and provide services to one another. Fundamentally, there is a tension between process isolation, which seeks to guarantee that a process cannot corrupt the code or data of another process, and inter-process communication, which is charged with providing a channel for data exchange. If a system isolates processes weakly, then

communication is straightforward, for example, by allowing processes to directly read from and write to each other's memory. By contrast, strongly isolated processes may only communicate in more complicated and performance intensive ways, for example, using remote procedure calls (RPCs). Because of the performance considerations, modern operating systems favor ease of inter-process communication at the expense of strong process isolation. Hunt et al. refer to the current environment as the *open process architecture*, noting that modern systems support processes that can dynamically load libraries, generate code at run-time, share memory with other processes, and make use of a kernel API that allows direct modification of another process' state [73].

The open process architecture encourages tight coupling of software components, such as extensions to applications (e.g., web browser plug-ins) or to the system itself (e.g., device drivers). This leads to reliability concerns, as an untrusted component can crash the process or even the entire system that hosts it. Furthermore, malware is granted more access to available system resources, due to ambient authority. Sandboxing technologies and new system architectures have been proposed to solve these problems.

Sandboxing

A sandbox is an isolated environment inside which untrusted code can safely run without affecting the rest of the system. A sandbox provides a monitored interface through which the isolated code can request access to any resources outside of the sandbox that it might need in order to function. There are a variety of ways a sandbox can be implemented, which are discussed next. Note that this classification is based on that

described in [197].

Managed Code Environment. Programs written in certain languages, such as Java or C#, are compiled into an intermediate representation (bytecode) that, instead of executing in the native environment, target a virtual machine. These managed code environments, such as the Java Virtual Machine (JVM) and the Common Language Runtime (CLR), provide a sandbox in which application code is interpreted and monitored in order to support type safety and memory safety. The virtual machine prevents code from leaving the sandbox by disallowing direct access to pointers and automatically managing memory.

Software-Based Fault Isolation. Software-based fault isolation (SFI) enforces memory access constraints on sandboxed code in software, instead of using the facilities of memory management hardware. The advantage is that costly protection domain crossings are avoided, while allowing untrusted code (e.g., an extension) to safely run in the address space of a host process. To isolate a component, the compiler inserts run-time checks before load, store, and control instructions that confine memory access to a specific region within the host address space. The original SFI proposal targeted RISC processors [200]. PittSFIeld overcame limitations of the classic approach in order to implement the sandbox on the x86 CPU [120]. BGI is a recent proposal to isolate kernel extensions with low overhead on commodity hardware [20].

Hardware Isolation. Recall that the two hardware features that are readily utilized by modern operating systems to support isolation are protection domains and virtual memory. On commodity hardware, two approaches are taken to provide virtual memory: paging and segmentation. Paging-based systems divide memory into fixed-sized, contiguous sections (usually 4K bytes each) called pages. Segmentation approaches use variable length memory segments that are specified by base address, segment length, and access permissions as an entry in a local descriptor table (LDT). The x86 processor supports both paging and segmentation, with the former offering more flexibility, thus, being more readily used. Nevertheless, researchers have found a use for memory segments as isolation domains that can safely execute untrusted code in the context of a host application. For example, Vx32 makes use of the segmentation facilities on x86 hardware in combination with binary instrumentation to isolate untrusted components within a host process [46]. Native Client (NaCl) combines SFI and hardware segmentation in order to safely execute untrusted native code in the context of a web browser [212].

System Call Interposition. Modern operating systems use system calls as the mechanism by which a process interacts with other processes and resources on the system. Interposing on system calls provides the system with a mechanism to monitor a request, including its parameters, and to allow, reject, or modify the call, based on a policy. A number of approaches have been proposed that operate in user-mode, such as Janus [59] and MAPbox [1]. These implementations both make use of process tracing facilities available on the Solaris operating system to monitor system calls. Generic Software Wrappers is a kernel-mode approach for augmenting the security of general

COTS software [51]. Systrace is a sandbox for process confinement that allows for metered privilege elevation without the need for setuid programs [149].

Virtualization. The techniques described above use system call interposition in order to confine processes under a given policy. Interposition techniques can also be applied to virtualize a process' view of its environment. This is beneficial because untrusted processes can be given the impression that they are executing with their own set of resources, such as the file system and network stack, but they remain sandboxed from the rest of the system. At the operating system level, FreeBSD Jails [81] and Solaris Zones [148] are examples of this approach, as both methods monitor and redirect system calls in the kernel in order to isolate a process into a subtree of the file system. Windows User Account Control (UAC) virtualization serves a similar purpose. The mechanism virtualizes file system and registry requests by redirecting them to unprivileged locations [127].

A more heavyweight approach employs a virtual machine monitor [60] (VMM) to virtualize an entire operating system as a guest that runs on top of a native OS host. The VMM interposes on resource requests that are made in the guest system and translates them in order to be serviced by the underlying host OS. The goal is convince processes that run in the guest OS that they are executing on a physical machine. Unfortunately, this goal is not always met [54]. Examples of VMM-based technologies are VMWare [187], Xen [6], and QEMU [12].

New Architectures

The previous section discussed a variety of sandboxing proposals that can be directly integrated into contemporary systems. This section describes ideas for rearchitecting the systems themselves in order to provide for process isolation. We start with a discussion of modern web browsers and how they can be viewed as enclosed operating environments themselves. This is followed by an examination of a prominent research operating system.

Modern Web Browsers. In the modern World Wide Web, many sites contain diverse applications which offer a rich set of features that go well beyond the display of static documents. As such, complex websites can be thought of as programs that execute in the context of a web-based operating system (the browser) that mediates their access to system resources. Hence, modern web browsers are incorporating techniques to support process isolation in order to maintain reliability and security.

Internet Explorer (IE) 8 introduced a multi-process architecture called Loosely-Coupled IE (LCIE) that allows the browser chrome process to be separated from individual tab processes [215]. Furthermore, the browser implements a sandboxing feature called Protected Mode IE (PMIE) [177], which makes use of the Mandatory Integrity Control (MIC) [126] access control policy that was added to Windows Vista.[2] PMIE ensures that all untrusted code is tagged as "Low Integrity" and is, therefore, given limited access to operating system resources.

[2]MIC is an access control model that implements the "no write up" policy from the Biba Integrity Model [15].

The Chromium web browser has a similar multi-process architecture [158], but, in addition to leveraging MIC, it has additional sandboxing features [7]. First, Chromium runs a tab process under a restricted access token in order to limit access rights. Second, it executes the process in a job object that restricts privileges and provides further isolation. Third, it uses an alternate desktop in order to prevent windows messages from being sent to or received from the sandbox.

Experimental browsers such as OP [61] and Gazelle [201] use process isolation techniques for slightly different goals. In addition to protecting the host operating system from untrusted code that executes in the browser, these approaches take care to fully isolate website principals from one another. Much like operating systems, these browsers ensure that inter-site communication is carefully mediated and policies are maintained, such as the same-origin policy.

Singularity. Singularity is an experimental operating system that re-examines the design choices of contemporary operating systems in order to solve the problems that plague these systems, such as security vulnerabilities, over-provisioned process interaction capabilities, and software failures that are caused by co-locating extensions with their host components [74]. Singularity has three features that are particularly relevant to process isolation: software isolated processes, contract-based communication channels, and fine-grained discretionary access control.

A software isolated process (SIP) is a process that is isolated by using language safety rules and static type checking [2]. Singularity is a microkernel-based operating system that is implemented in managed code. Thus, the safety constraints that are imposed by

the managed environment preclude the need for CPU memory management hardware to enforce process boundaries. As such, SIPs may safely execute in the same address space without incurring the overhead of virtual-to-physical address translations or kernel traps.

Singularity proposes a *sealed process architecture* [73], which prohibits all of the open process architecture features that were discussed in Section 2.2.2: dynamically loaded libraries, run-time code generation, shared memory, and an invasive kernel API. Such features inevitably lead to unreliable and insecure programs. Instead, processes in Singularity communicate via a message-based inter-process communication strategy called *channels* [40]. Channel communication is validated by statically verified contracts that fully describe the semantics of messages and ensure safe message passing.

Finally, Singularity introduces a fine-grained discretionary access control model [209]. Because SIPs execute in the sealed process architecture, the SIP itself becomes a first-class entity whose identity can be validated by application manifests that describe the code. Thus, a manifest and an application role are used to construct a fine-grained principal designation (subject) that is used in access control decisions. Due to this specificity, objects have rich ACLs that are pattern matched against the subject through a regular expression. This scheme allows access control in Singularity to be more precise than in the course-grained models which are used in modern operating systems.

2.2.3 Information Flow Tracking

Access controls and process isolation techniques can help to constrain the resources that a process may access, but they can do little to control how the data contained in those resources is propagated. For example, a process may be allowed to read the system password file to perform an operation, but these mechanisms cannot prevent that process from subsequently leaking the password file, either directly over the network or through a covert channel [107]. Information flow tracking is the analysis of how data is propagated through a system. At a high level, information flow tracking entails deciding which data is of interest and marking (or *tainting*) it with a label. As the tainted data is used in an application (e.g., a tainted data value is assigned to a variable), its label is propagated (*taint propagation*). A *taint source* is an entry point where an information flow tracking system assigns a label to interesting data (e.g., a sensitive file). A *taint sink* is an exit point where the system analyzes outgoing data (e.g., when sending a network packet) and takes some action if the outbound data is tainted. This section describes approaches to track information flow through a system, both on a single host and in a distributed setting.

Host-Based Tracking. Many systems perform information flow (or taint) tracking at the host level. For example, experimental systems such as Asbestos [36] and HiStar [216] promote information flow to a first-class operating system abstraction in order to support precisely specified information flow policies. SubOS [76] augments the traditional Unix access control mechanism with the concept of a sub-process in order to reduce the privileges of processes that are tainted by interacting with untrusted files.

BackTracker [93] is an approach that builds explicit graphs that provide information about the interactions between resources and processes, as well as the order in which these interactions have occurred. Another system, process coloring [78], uses sets of labels (or "colors") to track resource interactions. In this case, label sets implicitly encode previous interactions, and the order between operations is lost. Finally, TQAna [37] and Panorama [213] perform dynamic taint analysis in order to track information flow for the detection of malicious data exfiltration by spyware.

Distributed Tracking. Building upon information flow tracking on an individual host, researchers have proposed ways to generalize this technique to a distributed setting. For example, in [95], the authors of BackTracker introduce a distributed version that connects processes that interact via the network. Similarly, DStar [217] was introduced as a distributed version of HiStar that can propagate label sets between components and processes that run on different hosts. Another system, Neon [219], colors data with byte-level "tints" and tracks these labels through the operating system and network. Finally, in [155, 157], the authors describe Pedigree, a system for distributed information flow tracking. Pedigree is similar to the previous systems in that it uses taint sets to record interactions between processes and resources, and it attaches these taint sets to network packets in order to exchange information between hosts. The system is different in that it proposes to make use of taint sets at the network level. For example, a firewall could inspect a taint set and determine that a connection was initiated by a process that had previously read a sensitive file.

2.3 Process Integrity Tracking

Maintaining the integrity of an executable image file on disk is relatively straightforward. For example, Tripwire [90] is a data integrity tool that periodically scans files on disk, computes a cryptographic hash of a file's contents, and consults a precomputed lookup table to determine if the hash result matches the expected value. If the new hash result does not match the corresponding value in the table, an alert is issued. Asserting the integrity of a process in execution is more challenging, for a number of reasons. First, a process can (and often does) dynamically load libraries into its address space during run-time. Second, it is not uncommon for a program to generate code on-the-fly, as is the case with code produced by a just-in-time (JIT) compiler. Finally, malware can inject its code into a process at any time (e.g., using Windows API functions such as `WriteProcessMemory`). This section describes some of the approaches that are related to the goal of ensuring process integrity.

2.3.1 Kernel Integrity Checking

Modern operating systems consist of a monolithic kernel design that includes a large number of executable modules that interact in complex ways. Furthermore, since all components in the kernel share the same address space, a vulnerability in one component could subvert the integrity of the entire operating system. The primary line of defense against attacks on the kernel are the hardware supported isolation primitives discussed in Section 2.2.2: virtual memory protection, which disallows user-mode code to directly access the address space of the kernel, and protection domain sepa-

ration, which requires processes to invoke a kernel trap through a standard interface in order to make privileged requests. As alluded to above, these defenses fail when a vulnerability exists in a kernel component. The following approaches seek to ensure the integrity of the kernel, in spite of these issues.

A number of research efforts operate on the principle that analyzing the kernel from multiple perspectives can help determine violations of kernel integrity. Strider GhostBuster [203] and RootkitRevealer [24] monitor the system from two vantage points by using high-level API calls and low-level direct kernel observation. These systems raise an alert if a discrepancy is detected after querying for the same system information from both points of view. Similarly, VMwatcher [79] uses a virtual memory monitor (VMM) to compare an "in-the-box" to an "out-of-the-box" representation of kernel state. Patagonix [112] implements an approach for "lie detection" in a hypervisor by tracking all processes that are executing on the host and comparing its list to that which is produced by running traditional reporting utilities, such as `ps` or `Taskmgr.exe`, in the guest OS. Finally, Copilot [141] uses a PCI add-in card to obtain a snapshot of kernel memory. The image is then analyzed offline and possible kernel integrity violations are detected by comparing MD5 hashes of the captured kernel modules to known-good versions from an assumed correct state.

Other systems modify the memory or divert the control flow of kernel code in order to detect and prevent malicious intrusion. Livewire [56] is a host-based intrusion detection system that is built into a VMM. The system monitors the guest kernel state to detect intrusions, and it protects kernel integrity by marking code segments as read-only. NICKLE [159] is a lightweight VMM-based system that prevents unauthorized

execution by using a memory shadowing technique to store authenticated kernel code in shadow memory and by diverting execution to this memory. Finally, HookSafe [204] finds all potential hook points in the kernel and uses a hypervisor to impose a thin hook indirection layer to relocate the hooks to safe storage in order to prevent them from being hijacked.

2.3.2 Trusted Computing

Trusted computing is a technology that has been developed with the goal of maintaining the integrity of an entire computing platform in order to establish a notion of trust. Modern general-purpose operating environments can be considered *open platforms* because they allow users to install a variety of software and carry out a diverse set of tasks (e.g., play games, send emails, and perform online banking transactions) without any assurance with respect to the integrity of the operating system kernel. This is in contrast to *closed platforms*, such as set-top boxes, gaming consoles, and automated teller machines (ATMs), which maintain a strict set of special-purpose, tamper-resistant components. Closed platforms can offer integrity assurances, but they are too inflexible for general computing needs. Trusted computing attempts to bridge the gap with a *trusted platform* that allows diverse applications and workflows while maintaining platform integrity.

Code Identity

In order to provide for trusted computing, a platform must be able to convey information about its current state. This state encapsulates the hardware configuration of the host as well as the code that it has executed. The hardware configuration remains relatively static and can be validated by the manufacturer, for example, via a signed certificate, but the code that runs on a computer varies wildly over time and is difficult to identify. To this end, trusted computing platforms introduce the concept of *code identity* to describe executing software components (applications, kernel extensions, etc).

Code identity is typically established by taking a *measurement*, that is, computing a cryptographic hash over a component's executable image, loaded libraries, and inputs (e.g., command line arguments and configuration files). Since code can change over time (e.g., due to dynamic code generation), the measurement is taken just before the component begins execution. This ensures that the measurement has a global meaning, that is, the identity of a component will be the same across platforms.

Combining the hardware configuration with the measurements that identify all executing code establishes a platform state. However, it remains a challenge to ensure the integrity of the entity that performs these measurements. For user-mode application code, a component in the operating system kernel would be a logical choice, since the kernel is responsible for launching user-mode processes, and it resides in a privileged protection domain (Ring 0). Unfortunately, the kernel itself is vulnerable to attack (recall Section 1.1.2), so a malicious extension could be deployed to falsify the identity

measurements. Therefore, to maintain the integrity of the kernel, the boot loader takes a measurement of the operating system when it is loaded. Recursively, the BIOS must measure the boot loader before passing control to it. This forms a *chain of trust* that links the hardware and software components that are responsible for bringing the system from its initial boot configuration to the current state. To begin the chain, code identity requires a tamper-proof *root of trust* that can be depended upon to make the first measurement. The Trusted Platform Module (TPM) coprocessor has been developed to fulfill this role [62].

Using Measurements in a Local Context

Secure Boot. One important application of the measurements taken by the chain of trust is *secure boot* [57]. Secure boot is a mechanism to provide assurance that the system booted into a trusted state. This is accomplished by each component in the chain of trust comparing its measurement against a list of authorized components that is provided by a trusted authority. The boot process is aborted if a measurement is not authorized, therefore a successful boot indicates an assurance of platform integrity. AEGIS [4] is a system that implements secure boot. In AEGIS, when a component first executes, its identity is matched against a certificate that enumerates permitted software. The certificate is supplied by the owner of the platform. If the component does not have an appropriate certificate, it will not execute.

Sealed Storage. Code identity provides the system with a set of primitives upon which fine-grain access controls can be built. These controls alleviate some of the

issues that were discussed in Section 2.2.1 concerning ambient authority. Recall that a major problem with access control in contemporary operating systems is user-based authorization. Code identity measurements provide a means for an application to precisely specify the entities that are able to access the files (or *secrets*) it creates, through a mechanism called *sealed storage*. When using sealed storage, a process submits to the TPM a secret and a measurement that represents the subject that should be given access to the secret. The TPM *seals* the secret to this measurement by encrypting the secret under a key that is based on the measurement and other platform configuration information. Therefore, the secret can only be *unsealed* by the intended recipient on the same platform. This property can help prevent the leaking of secret data; for example, Microsoft's BitLocker feature [124] for full disk encryption uses sealed storage in its implementation on appropriate hardware. However, sealed storage has also incited controversy due its potential application for digital rights management (DRM) enforcement [3].

Using Measurements in a Remote Context

Remote Attestation. While secure boot and sealed storage help to ensure trust on the local host, trusted computing provides *remote attestation* to establish trust with respect to remote parties. Consider, as an example, a user that is doing online banking. The banking website would like an assurance that it is communicating with the actual user and not malware acting on the user's behalf or a miscreant that has stolen the user's credentials and has logged in from a different location. Remote attestation can be used to assure a remote party that a host is operating in a trusted state. The goal

of remote attestation is to convince a remote party (called a *verifier*) that the platform (in this context, called an *attestor*) is operating in a trusted state. The attestor sends a list of integrity measurements to the verifier to demonstrate that it is in a trusted state. The verifier then checks the measurements against its understanding of what the attestor's configuration should be and certifies the attestor if the configurations match. In the banking example above, after certification, the verifier could provide the attestor and the banking website with a session key over which to establish a trusted banking session.

A number of systems have been proposed to provide for remote attestation. Sailer et al. augment the Linux platform to use a TPM for attestation [166]. Terra [55] uses a trusted VMM to partition the host operating system into open and trusted platforms that can operate side-by-side. Because the virtual trusted platform is isolated by design, hardware attestation is only needed to attest that the trusted VMM is executing. Flicker [121] is an attestation infrastructure that aims to minimize the trusted computing base (TCB) while providing strong isolation guarantees for running code. Finally, Pioneer [170] is a software-based approach to remote attestation. Under certain assumptions, Pioneer allows a verifier to obtain strong assurance that the code which is executed on an untrusted remote platform has not been tampered with. However, Pioneer supports the trustworthy execution of code on an untrusted platform, and, in order to achieve this hard-to-obtain goal, it imposes substantial restrictions on the remote system that executes the code and on the associated network environment.

Chapter 3

Analyzing Malware by Orchestrating a Botnet Takeover

As we discussed in Section 2.1, there are a number of techniques that can be employed to observe the behavior and activities of bots and botnets, all of which fall under the rubric of malware analysis. While analyzing a bot sample in an instrumented environment can assist a researcher in classifying the malware, such analysis may be incomplete (e.g., it may miss trigger-based behavior). Furthermore, laboratory analysis of an isolated bot instance cannot answer material questions regarding active and dangerous botnet deployments. For example, it is important to know how large the active population is of a live botnet, which attacks are actively being leveraged, and what is the impact of these attacks. Such information is necessary to make decisions regarding how resources should be prioritized and deployed to defend against botnets in the wild.

In this chapter, we describe our experience with actively seizing control of the Torpig botnet for ten days. [1] Torpig, which has been described in [176] as "one of the most advanced pieces of crimeware ever created," is a type of malware that is typically associated with bank account and credit card theft. However, as we will describe in detail, it also steals a variety of other personal information.

3.1 Torpig Background

Torpig is a malicious bot that utilizes sophisticated techniques to locate sensitive information on the victim computers on which it runs. The bot then periodically exfiltrates the stolen data back to its controllers. Torpig is distributed as a component of Mebroot, an advanced rootkit that takes control of a machine by replacing the system's Master Boot Record (MBR) [89]. Figure 3.1 illustrates the botnet's infrastructure and workflow, which we will discuss next.

Victims become infected by Mebroot through drive-by download attacks [150]. In these attacks, web pages on legitimate but vulnerable websites (step ❶ in Figure 3.1) are modified to include HTML tags that cause the victim's browser to request JavaScript code (❷) from a website (the *drive-by download server*) under control of the attackers (❸). This JavaScript code launches a number of exploits against the browser or some of its components, such as ActiveX controls and plug-ins. If any exploit is successful, an executable is downloaded from the drive-by download server to the victim machine, and it is executed (❹).

[1] Earlier versions of this work were published in [185] and [186].

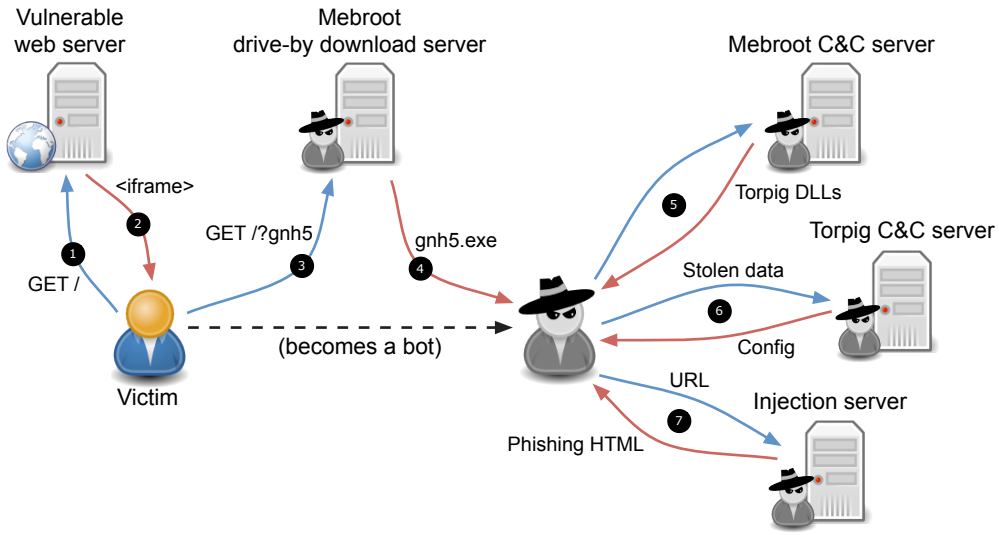


Figure 3.1: The Torpig network infrastructure.

The downloaded executable acts as an installer for Mebroot. The installer injects a DLL into the file manager process (`Explorer.exe`), and execution continues in the file manager's context. This makes all subsequent actions appear as if they were performed by a legitimate system process. The installer then loads a kernel driver that wraps the original disk driver (`Disk.sys`). At this point, the installer has raw disk access on the infected machine. The installer can then overwrite the MBR of the system with Mebroot. After a few minutes, the machine automatically reboots, and Mebroot is loaded from the MBR.

Mebroot has no malicious capability in and of itself. Instead, it provides a generic platform that other modules can leverage to perform their malicious actions. In particular, Mebroot provides functionality to manage (i.e., install, uninstall, and activate) these additional modules. Immediately after the initial reboot, Mebroot contacts the *Mebroot C&C server* to obtain malicious modules (❺). During the time that we moni-

tored Mebroot, the C&C server distributed three modules, which comprise the Torpig malware. Mebroot injects these modules (DLLs) into a number of applications. These applications include the Service Control Manager (`Services.exe`), the file manager, and 29 other popular applications, such as web browsers (e.g., Internet Explorer, Firefox, Opera), FTP clients (CuteFTP, LeechFTP), email clients (e.g., Thunderbird, Outlook, Eudora), instant messengers (e.g., Skype, ICQ), and system programs (e.g., the command line interpreter, `Cmd.exe`).

After the injection, Torpig can inspect the data that is handled by these programs to identify and store interesting pieces of information, such as credentials for online accounts and stored passwords. Periodically (every twenty minutes, during the time we monitored the botnet), Torpig contacts the *Torpig C&C server* to upload the data that was stolen since the previous reporting time (❻). All communication with the server is made over HTTP and is protected by a simple obfuscation mechanism, which was broken by security researchers at the end of 2008 [77]. The C&C server can reply to a bot in one of several ways. The server can simply acknowledge the data. We call this reply an *okn* response, as this string is contained in the server's reply. In addition, the C&C server can send a configuration file to the bot (we call this reply an *okc* response). The configuration file is obfuscated using a simple XOR-11 encoding. It specifies how often the bot should contact the C&C server, a set of hard-coded servers to be used as backup, and a set of parameters with which to perform *man-in-the-browser* phishing attacks [65].

Torpig uses phishing attacks to actively elicit additional, sensitive information from its victims, which, otherwise, may not be observed during the passive monitoring that

it normally performs. These attacks occur in two steps. First, whenever the infected victim visits one of the domains that is specified in the configuration file (typically, a banking website), Torpig issues a request to an *injection server*. The server's response specifies a page on the target domain where the attack should be triggered (we call this page the *trigger page*, and it is typically set to the login page of a site), a URL on the injection server that contains the phishing content (the *injection URL*), and a number of parameters that are used to configure the attack (e.g., whether the attack is active and the maximum number of times that it can be launched). The second step occurs when the user visits the trigger page. At this time, Torpig requests the injection URL from the injection server and injects the returned content into the user's browser (❼). This content typically consists of an HTML form that is designed to dupe the user into divulging sensitive information, for example, credit card numbers and social security numbers.

These phishing attacks are very difficult to detect, even for attentive users. In fact, the injected content carefully reproduces the style of the target website. Furthermore, the injection mechanism defies all phishing indicators that are included in modern browsers. For example, the SSL configuration appears correct, and so does the URL that is displayed in the address bar. An example screenshot of a Torpig phishing page for Wells Fargo Bank is shown in Figure 3.2. Notice that the URL points to the correct domain, the SSL certificate has been validated, and the address bar displays a padlock. Also, the page has the same style as others on the original website.

In summary, Torpig relies on a complex network infrastructure to infect machines, retrieve updates, perform active phishing attacks, and send the stolen information to its

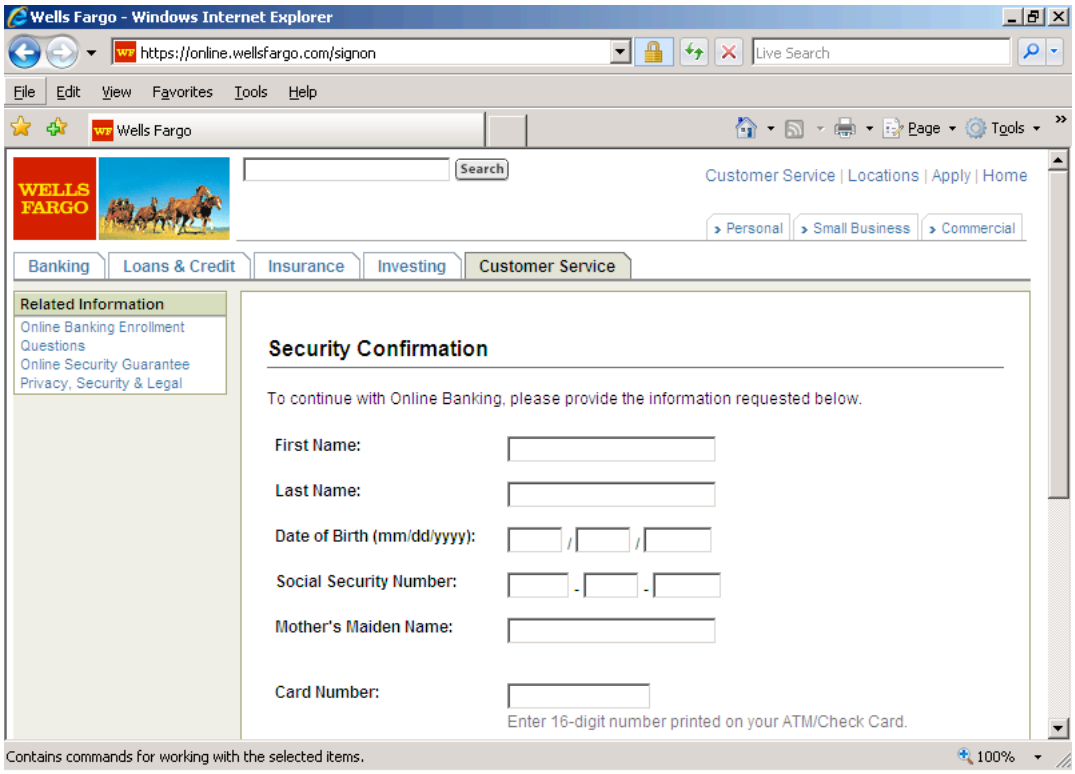


Figure 3.2: A man-in-the-browser phishing attack.

controllers. However, we observed that the schemes that are used to protect the communication in the Torpig botnet are insufficient to guarantee basic security properties (confidentiality, integrity, and authenticity). This was a weakness that enabled us to seize control of the botnet.

3.2 Torpig Takeover

We were able to orchestrate the takeover of the Torpig botnet for a period of ten days from January 25th to February 4th, 2009. This was made possible due to a weakness

that we exploited in the botnet's C&C communication technique. In this section, we describe this technique in detail as well as how we came to take advantage of it.

3.2.1 Domain Flux Overview

A fundamental aspect of any botnet is that of coordination, that is, how the bots identify and communicate with their C&C servers. Traditionally, C&C hosts have been located by their bots using the servers' IP addresses, DNS names, or their node IDs in peer-to-peer overlays. In the recent past, botnet authors have identified several ways to make these schemes more flexible and robust against take-down actions, for example, by using IP fast flux techniques [70]. Using fast flux, the bots make a DNS request against a certain domain name that is mapped onto a set of IP addresses, which change frequently. This makes it more difficult to take down or block a specific C&C server. However, fast flux uses only a single domain name, which constitutes a single point of failure.

Torpig addresses this issue by using a different technique for locating its C&C servers, which we refer to as *domain flux*. Using domain flux, each bot employs a domain generation algorithm (DGA) to compute a list of domain names. This list is computed independently by each bot and is regenerated periodically. The bot then attempts to contact the hosts in the domain list in order until one succeeds, that is, until the domain name resolves to an IP address and the corresponding server provides a response that is valid in the botnet's protocol. If a domain is blocked (e.g., the registrar suspends it to comply with a take-down request), the bot simply uses the next domain in the

list. Domain flux is also used to contact the Mebroot C&C servers and the drive-by download servers. Domain flux is increasingly popular among botnet authors. In fact, similar mechanisms have been used by the Kraken [161] and Srizbi bots [210], and, more recently, by the Conficker worm [147].

Torpig Domain Generation Algorithm

Torpig's DGA takes two values as input: the current date and an integer value that was constant during our monitoring period. The algorithm first computes a "weekly" second-level domain (SLD) name, say *dw*, which depends on the current week and year, but is independent of the current day (i.e., it remains constant for the entire week). Using the generated SLD, *dw*, a bot appends a number of top-level domains (TLDs): *dw*`.com`, *dw*`.net`, and *dw*`.biz` (in that order). It then resolves each domain name in sequence and uses the resulting IP address in an attempt to connect to its C&C server. If all three connections fail, Torpig computes a "daily" SLD, say *dd*, which additionally depends on the current day (i.e., a new SLD, *dd*, is generated each day). As before, *dd*`.com` is tried first, followed by *dd*`.net` and *dd*`.biz`. If connections to these domains also fail, Torpig attempts to contact the domains that are hard-coded in its configuration file.

From a practical standpoint, domain flux generates a list of "rendezvous points" that may be used by the botmasters to control their bots. Not all of the domains that are generated by a DGA need to be valid for the botnet to be operative. However, there are two requirements that botmasters must satisfy in order to maintain control of the bot-

net. First, botmasters must control at least one of the domains that will be contacted by the bots. Second, botmasters must use mechanisms to prevent other groups from seizing domains that will be contacted by bots before the domains under the botmasters' control.

In practice, the Torpig controllers registered the weekly `.com` domain and, in a few cases, the corresponding `.net` domain (for backup purposes). There were two weaknesses to their approach. First, all future weekly and daily domain names were generated deterministically and could be calculated offline ahead of time. Second, the botmasters did not register all of the weekly domains in advance, which was a critical factor in enabling our takeover.

Implications of Domain Flux

The use of domain flux has important consequences in the arms race between botmasters and defenders. From the attacker's point of view, domain flux is yet another technique to potentially improve the resilience of the botnet against take-down attempts. More precisely, in the event that the current rendezvous point is taken down, the botmasters simply have to register the next domain in the domain list to regain control of their botnet. On the contrary, to the defender's advantage, domain flux opens up the possibility of sinkholing a botnet, by registering an available domain that is generated by the botnet's DGAs and returning an answer that is a valid C&C response (to keep bots from switching over to the next domain in the domain list). Torpig allowed both of these actions: C&C domain names were available for registration, and it was possible

to forge valid C&C responses.

The feasibility of sinkholing a botnet depends not only on technical means (e.g., the ability to reverse engineer the botnet protocol and to forge a valid C&C server's response), but also on economic factors, in particular, on the cost of registering a number of domains sufficient to make the takeover effective. Since domain registration comes at a price (currently, from about five to ten dollars per year per `.com` or `.net` domain name), botmasters could prevent attacks against domain flux by making them economically infeasible, for example, by forcing defenders to register a disproportionate number of domains. Unfortunately, this is a countermeasure that is already in use. Newer variants of Conficker generate 50,000 domains per day and introduce non-determinism in the generation algorithm [147]. Taking over all of the domains that are generated by Conficker would cost between $91.3 million and $182.5 million per year. Furthermore, the domain flux arms race is clearly in favor of the malware authors. Generating thousands more domains requires a simple modification to the bot code base, while registering them requires a significant investment of time and money.

In short, the idea of combating domain flux by simply acquiring more domains is clearly not scalable in the long term, and new approaches are needed to thwart the activities of botmasters. In particular, the security community should build a stronger relationship with registrars. Registrars, in fact, are the entity best positioned to mitigate malware that relies on DNS (including domain flux), but they often lack the resources, incentives, or culture to address problematic domain names that are registered for malicious purposes. Nevertheless, such relationships are beginning to take shape, as evidenced by several recent collaborative take-down operations [29, 17, 195]. In addition,

rogue registrars (those known to be a safe haven for the activity of cyber-criminals) should lose their accreditation. While processes exist to terminate registrar accreditation agreements (a recent case involved the infamous EstDomains registrar [18]), such mechanisms should be streamlined and used more promptly.

3.2.2 Taking Control of Torpig

As mentioned above, the Torpig botmasters neglected to register in advance many of the future weekly domains that the bots were to resolve in order to contact their C&C server. This was a vulnerability in their communication technique that we were able to exploit to take control of the botnet. To this end, we registered the `.com` and `.net` domains that were to be used by the botnet for three consecutive weeks, from January 25th to February 15th, 2009, and then we configured a server to which the bots would connect for C&C. However, on February 4th, the Mebroot controllers distributed a new Torpig binary with an updated domain generation algorithm. This ended our control prematurely after ten days. The Mebroot C&C server, in fact, allows botmasters to upgrade, remove, and install new malware components at any time, and are tightly controlled by the criminals. It is unclear why the controllers of the Mebroot botnet did not update the Torpig DGA sooner to thwart our sinkholing.

We configured the domain names that we registered so that they resolved to the IP addresses of two servers that were under our control (for redundancy). The servers were configured to receive and log all incoming HTTP requests and other network traffic. During the ten days that we controlled Torpig, we collected over 8.7GB of

Apache log files and 69GB of packet capture (pcap) data.

During our collection process, our servers were effectively functioning as the C&C hosts of the Torpig botnet. As such, we were very careful with the information that we gathered and with the commands that we provided to infected hosts. While operating our C&C servers, we followed previously established legal and ethical principles [19]. In particular, we protected the victims according to the following:

Principle 1. *The sinkholed botnet should be operated so that any harm and/or damage to victims and targets of attacks would be minimized.*

Principle 2. *The sinkholed botnet should collect enough information to enable notification and remediation of affected parties.*

We took several preventative measures to ensure Principle 1. In particular, when a bot contacted our server, we always replied with an okn message and never sent a new configuration file to the bot. This forced the bots to remain in contact only with our servers. If we had not replied with a valid Torpig response, the bots would have switched over to the `.biz` domains, which had already been registered by the botmasters. Although we could have replied by sending a blank configuration file to potentially remove the websites that were targeted by Torpig, we did not. We took this precaution to avoid unforeseen consequences (e.g., changing the behavior of the malware on critical computer systems). [2] Furthermore, we did not send a configuration file with a different HTML injection server IP address for the same reasons. We also periodically removed sensitive data from our C&C server so that the risk of a data breach was minimized.

[2] We note that recent take-down efforts by law enforcement have, indeed, explored the efficacy of sending bots a so called "kill" command that would incapacitate them [99, 195].

To notify the affected institutions and victims, we stored all data that was sent to us, in accordance with Principle 2. We worked with ISPs and law enforcement agencies, including the United States Department of Defense and FBI Cyber Crime units, to assist us with this effort.

3.3 Torpig Analysis

As mentioned previously, we collected almost 70GB of data over a period of ten days. The wealth of information that is contained in this data set is remarkable, including numerous passwords, credit card numbers, and bank account information, among other sensitive data. In this section, we present the results of our data analysis. Furthermore, we discuss important insights into the size of botnets that we gleaned due to the unique perspective that our takeover granted us.

3.3.1 Data Collection

All Torpig bots communicate with the C&C server through HTTP POST requests, an example of which is shown in Figure 3.3. The URL that is used for this request contains the hexadecimal representation of a *bot identifier* and a *submission header*. A bot identifier is a token that is computed on the basis of hardware and software characteristics of the infected machine. It is used as a symmetric decryption key for the submission header and POST body, [3] and it is sent in the clear.

[3]The encryption algorithm is a simple cipher that makes use of XOR and Base64 encoding.

```
POST /A15078D49EBA4C4E/qxoT4B5uUFFqw6c35AKDY...at6E0AaCxQg6nIGA

ts=1232724990&ip=192.168.0.1:&sport=8109&hport=8108&os=5.1.2600&
    cn=United%20States&nid=A15078D49EBA4C4E&bld=gnh5&ver=229
```

Figure 3.3: Sample POST request made by a Torpig bot (top) and the corresponding, decrypted submission header (bottom).

The submission header consists of a number of key-value pairs that provide basic information about the bot. More precisely, the header contains a timestamp that corresponds to when the configuration file was last updated (`ts`), the IP address of the bot (`ip`), the port numbers of SOCKS and HTTP proxies that Torpig opens on the infected machine (`sport` and `hport`), the operating system version and locale (`os` and `cn`), the bot identifier (`nid`), and the build and version number of the bot binary (`bld` and `ver`). Figure 3.3 shows a sample of the submission header information that is sent by a Torpig bot.

The request body consists of zero or more *data items* of different types, which depend on the information that was stolen. Table 3.1 shows the different data types that we observed during our analysis. In particular, *mailbox account* items contain the configuration information for email accounts, that is, the email address that is associated with the mailbox and the credentials that are required to access the mailbox and to send emails from it. Torpig obtains this information from email clients, such as Outlook, Thunderbird, and Eudora. *Email* items consist of email addresses, which presumably can be used for spam purposes. According to [198], Torpig initially used spam emails to propagate, which may give another explanation for the botmasters' interest in email addresses. *Form data* items contain the content of HTML forms that are submitted via POST requests by the victim's browser. More precisely, Torpig collects the URL that

hosts the form, the URL that the form is submitted to, and the name, type, and value of all form fields. These data items frequently contain the usernames and passwords that are required to authenticate with websites. Note that credentials that are transmitted over HTTPS are not safe from Torpig, since the bot hooks appropriate library functions and can, therefore, access the credentials before they are encrypted by the SSL layer. *HTTP account*, *FTP account*, *POP account*, and *SMTP account* data types contain the credentials that are used to access websites, FTP, POP, and SMTP accounts, respectively. Torpig obtains this information by exploiting the password manager functionality that is provided by most web and email clients. SMTP account items also contain the source and destination addresses of emails that are sent via SMTP. Finally, the *Windows password* data type is used to transmit Windows passwords and other uncategorized data elements. Figure 3.4 shows a sample of the data items sent by a Torpig bot.

Table 3.1: Data items sent to our C&C server by Torpig bots.

Data Type	Number of Data Items
Mailbox account	54,090
Email	1,258,862
Form data	11,966,532
HTTP account	411,039
FTP account	12,307
POP account	415,206
SMTP account	100,472
Windows password	1,235,122

```
[gnh5_229]
[MSO2002-MSO2003:pop.smith.com:
    John Smith:john@smith.com]
[pop3://john:smith@pop.smith.com:110]
[smtp://:@smtp.smith.com:25]
```

```
[gnh5_229]
POST /accounts/LoginAuth
Host:  www.google.com
POST_FORM:
Email=test@gmail.com
Passwd=test
```

Figure 3.4: Sample data sent by a Torpig bot: a mailbox account data item on the left and a form data item on the right.

3.3.2 Botnet Size

In this section, we address the problem of determining the size of the Torpig botnet. More precisely, we will be referring to two definitions of a botnet's size as introduced by Rajab et al. [154]: the botnet's *footprint*, which indicates the aggregated total number of machines that have been compromised over time, and the botnet's *live population*, which denotes the number of compromised hosts that are simultaneously communicating with the C&C server at a given time.

Botnet size approximations are often debated, as estimates vary wildly and are commonly incorrect and alarmist, particularly when reported in the popular press [110, 122, 44, 142]. Several methods have been proposed in the past to correctly estimate the size of botnets [52, 30, 156, 153, 85, 152]. These approaches are modeled after the characteristics of the botnet under analysis and vary along different axes, depending on whether the researchers have access to direct traces of infected machines or must resort to indirect measurements, whether they have a complete or partial view of the infected population, and, finally, whether individual bots are identified by using a network-level identifier (e.g., an IP address) or an application-defined identifier. It is this last characteristic of the Torpig botnet that enabled us to precisely count its footprint and live

population.

Counting Methodology

In comparison to previous studies, the Torpig C&C architecture provides an advantageous perspective with which to measure the botnet's size. In fact, since we centrally and directly observed every infected machine that normally would have connected to the botmasters' server during the ten days that we controlled the botnet, we had a complete view of all Torpig bots. In addition, our collection methodology was entirely passive and, thus, it avoided the problem of active probing that may have otherwise polluted the network. Finally, Torpig generates and transmits unique and persistent tokens that allowed us to accurately identify each infected machine.

To wit, we found that a number of the values that were sent by each bot as part of the submission header depend on the particular configuration of the infected machine. Furthermore, these values remain constant over time, thus providing an accurate method for identifying each bot during our period of observation. In particular, to identify each Torpig bot, we used a tuple of values that correspond to the following keys from the submission header: (`nid`, `os`, `cn`, `bld`, `ver`). By reverse engineering the Torpig binary, we determined that the value of the `nid` field is computed on the basis of the specific hardware and software characteristics of the infected machine's hard disk (including its model and serial numbers) and is therefore highly distinctive. However, we discovered that the value was not strictly unique to each machine; there were 2,079 cases in which the same `nid` appeared to be assigned to different machines, based on

their geographical location and other characteristics. Thus, to minimize the effect of these collisions, we also took into account the `os` (OS version number) and `cn` (locale) fields, which do not change unless the user changes operating systems or modifies the computer's locale information, and the `bld` and `ver` fields, which are derived from hard-coded values in the Torpig binary.

Several fields in the submission header were not used to construct the identifier tuple. In particular, we did not utilize the `ts` field (the timestamp of the configuration file) because its value is determined by the Torpig C&C server and not by any characteristics of the bot. We discarded the `ip` field since it may change depending on DHCP lease times and other network configurations. Finally, we ignored the `sport` and `hport` fields, which specify the proxy ports that Torpig opens on the local machine, because they may change upon each reboot of the infected machine.

Footprint and Live Population

By applying the identifier tuple to the network data that we captured during our takeover, we estimate that the botnet's footprint consisted of 182,914 infected hosts. From this count, we wanted to identify and remove hosts that correspond to security researchers and other individuals who probed our C&C server, since such hosts do not indicate actual victims of the botnet. We attributed requests to researchers if the URL contained `nid` values that were generated from standard configurations of virtual machines, such as VMWare and QEMU. [4] The rationale is that virtual machines are often used by an-

[4]The `nid` value depends on physical characteristics of the hard disk, and, by default, virtual machines provide virtual devices with a fixed model and serial number, making them easy to identify.

alysts to study malware in a controlled environment. We identified hosts as probers when they sent invalid requests, that is, requests that could not be generated by Torpig bots (e.g., HTTP GET requests). After using these heuristics to discount researchers and probers, we arrived at our final estimate of Torpig's footprint: 182,800 infected hosts.

We also used our counting methodology to estimate the size of Torpig's live population during the time that it was under our control. Specifically, we measured the median and average size of the live population to be 49,272 and 48,532 bots, respectively. The live population fluctuates periodically, with a peak around 9:00 a.m. (PST), when the most computers are simultaneously online in the United States and Europe. Conversely, we found that the live population was the smallest around 9:00 p.m. (PST). Figure 3.5 shows the number of new IP addresses that connected to our server on an hourly basis. The figure clearly demonstrates a diurnal pattern in the connections.

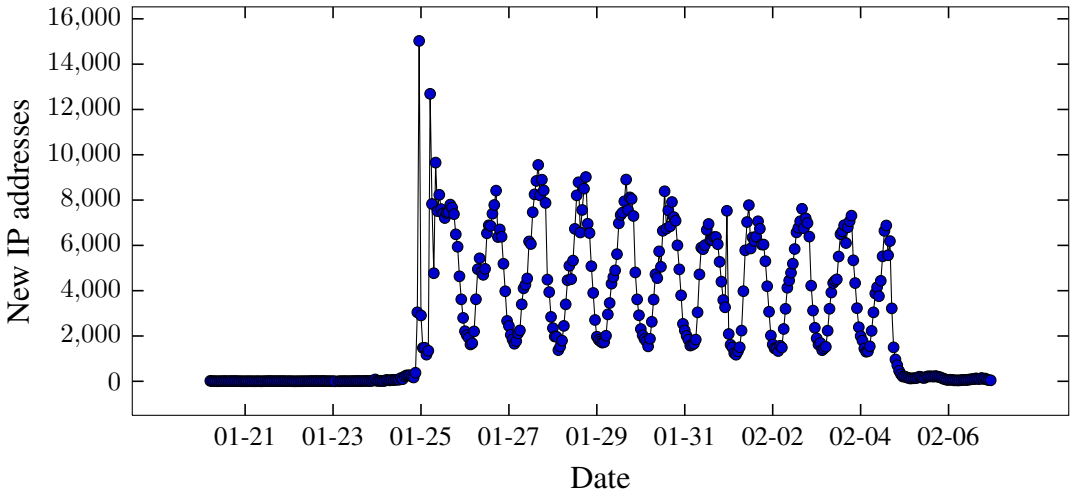


Figure 3.5: New unique bot IP addresses per hour.

It may seem that a more straightforward and general way to count the botnet's footprint and live population would be to enumerate the unique IP addresses that connect to C&C server. However, it is well-known that this simple counting metric is problematic, due to network effects such as DHCP churn and network address translation (NAT) [154]. To wit, while we observed 182,800 infected hosts using the identifier tuple counting metric, we enumerated 1,247,642 unique IP addresses over the course of our control of the botnet. Thus, if we were to simply count IP addresses to approximate the botnet's footprint, we would have overestimated its size by an order of magnitude. Furthermore, Figure 3.6 shows the number of new bots that contacted our server per hour, according to the identifier tuple counting metric. Clearly, there were significantly less new bots connecting over time than Figure 3.5 might lead one to believe.

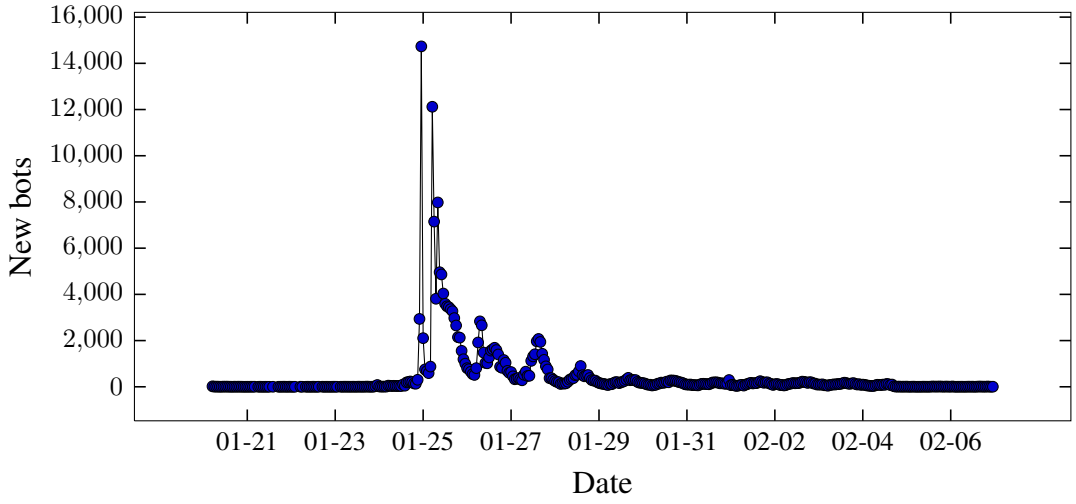


Figure 3.6: New bots per hour.

New Infections

The Torpig submission header provides a timestamp value (`ts`) that corresponds to when the bot most recently received a configuration file. We leveraged this fact to approximate the number of machines that were newly infected during our period of observation. This was accomplished by counting the number of distinct victims whose initial submission header contained a timestamp value of 0. Figure 3.7 shows the new infections over time. In total, we estimate that there were 49,294 new infections while the botnet was under our control. Note that new infections peaked on the 25th and the 27th of January. We can only speculate that a popular website was compromised on those days, and the site redirected its visitors to the drive-by download servers that were controlled by the botmasters.

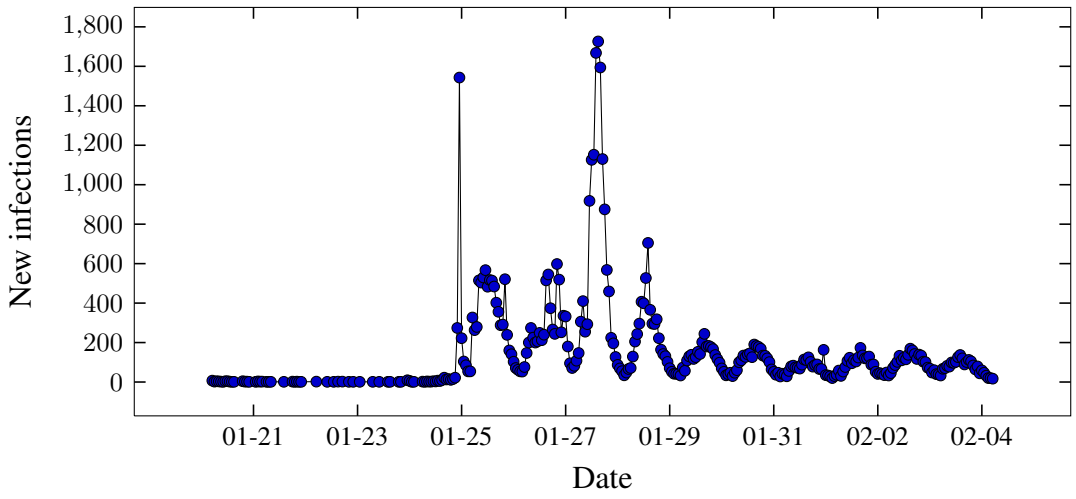


Figure 3.7: New Torpig infections over time.

3.3.3 Data Analysis

In this section, we discuss our analysis of the data that the Torpig bots stole from their victims and sent to our C&C server while the botnet was under our control. While we will focus on the financial data and password information that was actually stolen, we note that the botnet has the capability to do even greater harm. In particular, recall that Torpig bots open two ports on the victim host to be used as a SOCKS and HTTP proxy. These could be used for spamming, anonymous web navigation, or other dubious enterprises. Furthermore, the aggregate bandwidth of the bots, particularly those with high speed network access, enables the botnet to potentially leverage a formidable distributed denial-of-service (DDoS) attack.

Financial Data Theft

Torpig is specifically crafted to obtain information that can be readily monetized in the underground market. Financial credentials, such as bank account and credit card numbers, are particularly sought after. For example, the typical Torpig configuration file lists roughly 300 domains that belong to banks and other financial institutions that are the targets of the man-in-the-browser phishing attacks that were described in Section 3.1.

Table 3.2 reports the quantity of account numbers at financial institutions (e.g., banks, online trading, and investment companies) that were stolen by Torpig bots and sent to our C&C server. In ten days, Torpig obtained the credentials of 8,310 accounts at 410 different institutions. The most targeted institutions were PayPal (1,770 accounts),

Poste Italiane (765), Capital One (314), E*Trade (304), and Chase (217). By contrast, a large number of companies had only a handful of compromised accounts (e.g., 310 institutions had ten or less). Due to the large number of affected institutions, notifying all of the interested parties was a substantial undertaking. It is also interesting to observe that 38% of the credentials that were stolen by Torpig were obtained from the password manager of browsers, rather than by intercepting an actual login session. It was possible to infer that number because Torpig uses different data formats in which to upload stolen credentials from different sources.

Table 3.2: Financial institution account numbers that were stolen by Torpig.

Country	Number of Institutions	Number of Accounts
United States	60	4,287
Italy	34	1,459
Germany	122	641
Spain	18	228
Poland	14	102
Other	162	1,593
Total	410	8,310

Another target for collection by Torpig bots is credit card data. To mine credit card numbers from our data set, we used a card number validation heuristic that includes the Luhn algorithm and matching against the correct number of digits and numeric prefixes of card numbers from the most popular credit card companies. As a result, we extracted 1,660 unique credit and debit card numbers from our collected data. Through IP address geolocation, we surmise that 49% of the card numbers came from victims in the US, 12% from Italy, and 8% from Spain, with 40 other countries making up the balance. The most common cards include Visa (1,056 card numbers), MasterCard

(447), American Express (81), Maestro (36), and Discover (24).

While 86% of the victims contributed only a single card number, others offered a few more. Of particular interest is the case of a single victim from whom 30 credit card numbers were extracted. Upon manual examination, we discovered that the victim was an agent for an at-home, distributed call center. It seems that the card numbers were those of customers of the company that the agent was working for, and they were being entered into the call center's central database for order processing.

Quantifying the value of the financial information that was stolen by Torpig is an uncertain process because of the characteristics of the underground markets where it may end up being traded. A report by Symantec indicated approximate ranges of prices for common goods and, in particular, priced credit card numbers between \$0.10–\$25 and bank account numbers from \$10–\$1,000 [48]. If these figures are accurate, then in ten days of activity, the Torpig controllers may have profited anywhere from \$83K to \$8.3M.

Furthermore, we wanted to determine the rate at which the botnet produces *new* financial information for its controllers. Clearly, a botnet that generates all of its value in a few days and later only recycles stale information is less valuable than one in which fresh data is steadily produced. Figure 3.8 shows the rate at which new bank account and credit card numbers were obtained during our monitoring period. As the figure illustrates, in the ten days that we had control of the botnet, new data was continuously stolen and reported by Torpig bots.

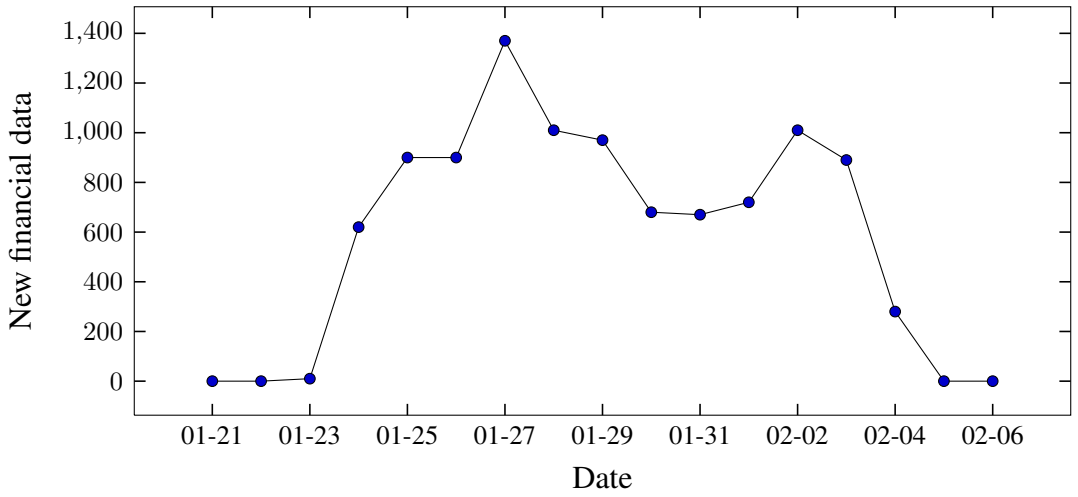


Figure 3.8: Arrival rate of financial data.

Password Analysis

It is commonly believed among security experts that a large percentage of Internet users routinely neglect the importance of using and maintaining strong passwords. A poll conducted by Sophos in March 2009 supports this claim, as it found that one-third of 676 users relied on weak passwords and reused authentication credentials across different web services [180].

Our data set allowed us to cross-validate these results. Torpig bots uploaded 297,962 unique credentials (username and password pairs) to our C&C server from 52,540 different infected machines. Note that our data set is two orders of magnitude larger than the one that was used in the Sophos poll, and it consists of credentials that were actually used in the wild, rather than relying on the feedback of poll respondents. Our analysis found that almost 28% of the victims reused their authentication credentials for accessing 368,501 websites.

In addition to checking for credential reuse, we also conducted an experiment to assess the strength of the 173,686 unique passwords that we discovered in the experiment above. Note that there is no standard, well-defined criteria to measure password strength, so we devised our own metric. To assess password strength, we encrypted the passwords that we mined from our data set, and we measured the time that it took for a password cracking tool to recover the plaintext of each password. The rationale for using this metric is that cracking tools can easily "guess" weak passwords, such as those that consist of dictionary words or those that contain simple patterns (e.g., a dictionary word followed by a number). To this end, we created a Unix-like password file from our password corpus, and we used the file as input to John the Ripper, [5] a popular password cracking tool. The results are presented in Figure 3.9.

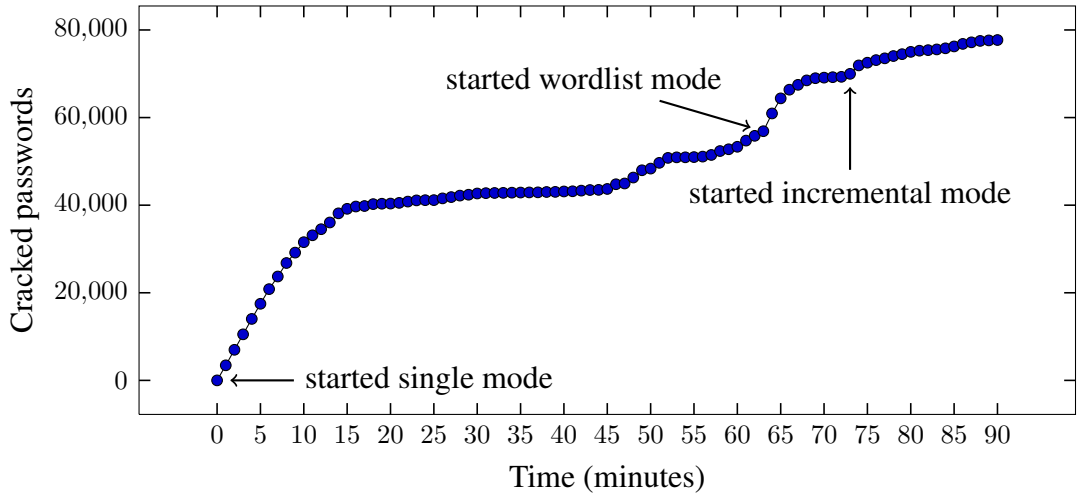


Figure 3.9: Number of cracked passwords over time.

The tool recovered approximately 56,000 passwords in less than 65 minutes by using permutation, substitution, and other simple replacement rules (John's "single" mode).

[5] http://www.openwall.com/john/

Another 14,000 passwords were recovered in the next 10 minutes when the password cracker switched to a dictionary attack ("wordlist") mode. Thus, in less than 75 minutes, more than 40% of the passwords were recovered. 30,000 additional passwords were recovered in the next 24 hours by brute force (John's "incremental" mode).

3.4 Summary

We presented a comprehensive analysis of the operations of the Torpig botnet. We gleaned a number of insights from the analysis of the 70GB of data that we collected, as well as from the process of obtaining (and losing) control of the botnet. First, we found that it is inaccurate to evaluate a botnet's size based simply on the count of distinct IP addresses of its bots. This is due to a variety of network effects, such as DHCP churn and usage of NAT devices, VPNs, proxies, and firewalls. This finding confirms previous, similar results and demonstrates that many reports of botnet infections are both overinflated and alarmist.

Second, the victims of botnets are often users with poorly maintained machines, and the users choose easily guessable passwords to protect access to sensitive websites. This is evidence that the rejection of security advice is fundamentally a cultural problem. Even though Internet users are educated and they understand concepts such as the physical security and the necessary maintenance of a car, they do not always understand the consequences of irresponsible behavior when using a computer. Therefore, in addition to novel tools and techniques to combat botnets and other forms of malware, it is necessary to better educate the Internet community so that the number of potential

victims is reduced.

Finally, we learned that interacting with registrars, hosting facilities, victim institutions, and law enforcement is a rather complicated process. We found that it is difficult to initially make contact with representatives of each of these organizations, but, once we were vetted, they can be quite helpful. For example, our contacts at the FBI were indispensable in fostering relationships with financial institutions so that we could repatriate the sensitive account information of their customers that was stolen by the Torpig bots.

Chapter 4

Detecting Malware by Tracking Dynamic Code Identity

The analysis that we described in the previous chapter demonstrates how dangerous malware (such as the Torpig bot) can be, once it infects a host. The potential for malware to do harm is intensified because modern operating systems implement user-based authorization for access control, thus giving processes the same access rights as the user account under which they run. This violates the *principle of least privilege* [168] because processes are implicitly given more access rights than they need, which is particularly problematic in the case of malware. For example, superfluous access rights allow Torpig to steal sensitive data from its victims. A more robust strategy to mitigate the effects of running malware is to make access control decisions based on the *identity* of the executing software. That is, instead of granting the same set of access rights to all applications that are run by a user, it would be beneficial to differentiate between

programs and to assign different rights based on their individual needs. For example, a security policy could enforce that only a particular (unmodified) word processing application should access a sensitive document, or an online banking application might refuse to carry out a transaction on behalf of a user unless it can identify that the user is executing a trusted web browser. An even stronger policy could define a set of trusted (whitelisted) applications, while the execution of any other code would be denied.

Enforcing fine-grained access control policies on an application basis requires a strong notion of *code identity* [143]. Code identity is a primitive that allows an entity (e.g., a security enforcement component) to recognize a known, trusted application as it executes. Code identity is the fundamental primitive that enables trusted computing mechanisms such as sealed storage [39] and remote attestation [166].

The state-of-the-art in implementing code identity involves taking *measurements* of a process by computing a cryptographic hash over the executable file, its load-time dependencies (libraries), and perhaps its configuration. The measurements are usually taken when a process is loaded, but just before it executes [143]. A measurement is computed at this time because it includes the contents of the entire executable file, which contains state that may change over the course of execution (e.g., the data segment). Taking a measurement after this state has been altered would make it difficult to assign a global meaning to the measurement (i.e., the code identity of the same application would appear to change).

Since the code identity primitive is fundamentally static, it fails to capture the true run-time identity of a process. Parno et al. acknowledge this limitation, and they

agree that this is problematic because it makes it possible to exploit a running process without an update to the identity [143]. For example, if an attacker is able to exploit a buffer overflow vulnerability and execute arbitrary code in the context of a process, no measurement will be taken and, thus, its code identity will be the same as if it had not been exploited.

In this chapter, we address the problem of static code identity, and we propose DYMO, a system that provides a *dynamic* code identity primitive that continuously tracks the run-time integrity of a process. [1] In particular, we introduce a host-based component that binds each process to an *identity label* that implements dynamic code identity by encapsulating all of the code that the process attempts to execute. More precisely, for each process, our system computes a cryptographic hash over each executable region in the process' address space. The individual hash values are collected and associated with the corresponding process. This yields an identity label that reflects the executable code that the application can run, including dynamic changes to code regions such as the addition of libraries that are loaded at run-time or code that is generated on-the-fly, for example, by a JIT compiler or an exploit that targets a memory vulnerability.

Identity labels have a variety of practical uses. For example, labels can be used in a host-based application whitelisting solution that can terminate processes when their run-time integrity is compromised (e.g., as the result of a drive-by download attack against a web browser). Also, identity labels can enable fine-grained access control policies such as only granting network access to specifically authorized programs (e.g., known web browsers and email clients).

[1] An earlier version of this work was published in [58].

To demonstrate how the use of identity labels can be extended into the network, we implemented an extension to DYMO that provides provenance information to all outgoing network connections. More precisely, we extended DYMO with a component that marks each TCP connection and UDP packet with a compressed identity label that corresponds to the application code that has generated the connection (or packet). This label is embedded in the network traffic at the IP layer, and, therefore, it can be easily inspected by both network devices and by the host that receives the traffic.

We have implemented our system as a kernel extension for Windows XP and tested it on several hardware platforms (a "bare metal" installation and two virtualized environments). Our experiments show that identity labels are the same when the same application is run on different systems. Moreover, when a malware program or an exploit attempts to inject code into a legitimate application, the label for this application is correctly updated.

4.1 System Overview

In this section, we first discuss the requirements for our identity labels in more detail. Then, we present an overview of DYMO, our system that implements these labels and provides dynamic code identity for processes.

4.1.1 System Requirements

A system that aims to provide dynamic code identity must fulfill three key requirements: First, identity labels must be *precise*. That is, a label must uniquely identify a running application. This implies that two different applications receive different labels. Moreover, it also means that a particular application receives the same label when executed multiple times on different hardware platforms or with slightly different dynamic libraries. This is crucial in order to write meaningful security polices that assign permissions on the basis of applications.

The second requirement is that identity labels must be *secure*. That is, it must be impossible (or very difficult) for a malicious process to assume the identity of a legitimate application. Otherwise, a malicious process can easily bypass any security enforcement mechanism that is based on code identity simply by impersonating an application that has the desired permissions.

The third requirement is that the implementation of the mechanism that computes identity labels must be *efficient*. Program execution on current operating systems is highly dynamic, and events in which a process adds additional code to its address space (typically in the form of dynamic libraries) are common. Also, the access permissions of code segments are changed surprisingly often. Thus, any mechanism that aims to maintain an up-to-date view of the running code will be invoked frequently, and, thus, must be fast.

4.1.2 System Design

To capture the dynamic identity of code, and to compute identity labels, we propose an approach that dynamically tracks all executable code regions in a process' address space. Typically, these code regions contain the instructions of the application code as well as the code sections of libraries, including those that are dynamically loaded. DYMO computes a cryptographic hash over the content of each code section, and it uses the set of hashes as the process' identity label.

Precise Label Computation. DYMO ensures the precision of identity labels, even in cases where an application loads slightly different sets of libraries on different executions. This can happen when applications load certain libraries only when the need arises, for example, when the user visits a web page that requires a particular browser plug-in. In such cases, two identity labels for two executions of the same application will contain an identical set of hashes for those libraries that are present in both processes, while one label will have extra hashes for any additional libraries that are loaded.

Typically, executable regions in a process' address space correspond to code sections of the binary or libraries. However, this is not always the case. For example, malicious processes can inject code into running applications (e.g., using Windows API functions such as `VirtualAllocEx` and `WriteProcessMemory`). In addition, when a legitimate application has a security vulnerability (such as a buffer overflow), it is possible to inject shellcode into the application, which alters its behavior. Our identity

labels encapsulate such code, because DYMO keeps track of all executable memory regions, independent of the way in which these regions were created.

Handling Dynamically Generated Code. An important difference from previous systems that compute hashes of code regions to establish code identity is that DYMO supports dynamically generated code. For this, one could simply choose to hash code regions that are dynamically created (similar to regular program code). Unfortunately, it is likely that such code regions change between program executions. For example, consider a just-in-time compiler for JavaScript that runs in a browser. Obviously, the code that is generated by this JIT compiler component depends on the web pages that the user visits. Thus, the hashes that are associated with dynamically generated code regions likely change very frequently. As a result, even though such a hash would precisely capture the generated code, its value is essentially meaningless. For this reason, we decided not to hash dynamic code regions directly. Instead, whenever there are dynamically created, executable memory regions, we add information to the label that reflects the generated code and the library that is responsible for it. The rationale is that we want to allow only certain known (and trusted) parts of the application code to dynamically generate instructions. However, there are no restrictions on the actual instructions that these regions can contain. While this opens a small window of opportunity for an attacker, a successful exploit requires one to find a vulnerability in a library that is permitted to generate code, and this vulnerability must be such that it allows one to inject data into executable memory regions that this library has previously allocated. This makes it very difficult for a malicious program or an attacker to coerce a legitimate program to execute unwanted code.

Secure Label Computation. Identity labels must be secure against forging. This requires that malicious processes cannot bypass or tamper with the component that computes these labels. In other words, DYMO must execute at a higher privilege than malicious code that may tamper with the label computation.

One possible way to implement DYMO is inside a virtual machine monitor (VMM). This makes it easy to argue that the component is protected from the guest OS and that it cannot be bypassed. Furthermore, it would be a convenient location to implement our extensions, since we could use an open-source VMM. Another way to implement DYMO is as part of the operating system kernel. In this case, one must assume that malicious processes only run with regular user (non-administrator) privileges. Moreover, this venue requires more implementation effort given that there is no source code available for Windows. However, on the upside, implementing DYMO as part of the operating system kernel makes real-world deployment much more feasible, since it does not require users to run an additional, trusted layer (such as a virtual machine) underneath the OS.

For this work, we invested a substantial effort to demonstrate that the system can be implemented as part of the Windows operating system. This was a deliberate design decision that makes DYMO easier to deploy. We also believe that it is reasonable to assume that the attacker does not have root privileges. With the latest releases of its OS, Microsoft is aggressively pushing towards a model where users are no longer authenticated as administrator but run as regular users [127]. Also, recent studies have shown that malware increasingly adapts to this situation and runs properly even without administrator privileges [9].

Efficient Label Computation. Computing labels for programs should only incur a small performance penalty. We add only a few instructions to the fast path in the Windows memory management routines (which are executed for every page fault). Moreover, the label computation is done incrementally; DYMO only needs to inspect the new, executable memory regions that are added to the process address space. As a result, our label computation is fast, as demonstrated by the performance overhead measured in our experiments (which are discussed in Section 4.4).

4.2 System Implementation

In this section, we describe DYMO's implementation in detail. In particular, we discuss how our system extends the Windows XP kernel to track the executable regions of a process and uses this information to compute identity labels.

Dynamically maintaining a process' identity over the course of its execution is a difficult problem. The first concern is that processes load dynamic link libraries (DLLs) during run-time, which makes it difficult to predetermine all of the code segments that will reside in a process' address space. Second, processes may allocate arbitrary memory regions with execute permission, for example, when dynamically generating code. This is commonly done by packed malware, which produces most of its code on-the-fly in an effort to thwart signature-based detection, but also by just-in-time compilers that generate code dynamically. A third issue concerns image rebasing. When the preferred load addresses of two DLLs conflict, one has to be relocated, and all addresses of functions and global variables must be patched in the code segment of the rebased DLL.

This poses a problem because we do not want the identities of two processes to differ simply because of differences in DLL load order. DYMO is able to track a process' identity in spite of these problems, as discussed in the following sections.

4.2.1 System Initialization

We assume that DYMO is installed on an uncompromised machine and is executed before any malicious process runs. Our system begins its operation by registering for kernel-provided callbacks that are associated with process creation and image loading (via `PsSetCreateProcessNotifyRoutine` and `PsSetLoadImageNotifyRoutine`, respectively) and hooking the NT kernel system services responsible for allocating memory, mapping files, and changing the protection of a memory region (these functions are `NtAllocateVirtualMemory`, `NtMapViewOfSection`, and `NtProtectVirtualMemory`, respectively).

By registering these callbacks and hooks, DYMO can observe and track all regions of memory from which a process could potentially execute code. DYMO also hooks the page fault handler so that it will be alerted when a tracked memory region has been requested for execution. This allows for the inclusion of this region into the identity label. This alert strategy makes use of hardware-enforced Data Execution Prevention (DEP/NX) [128]. DEP/NX utilizes No eXecute hardware support to prevent execute access on memory pages that have the NX bit set. Note that only those DEP/NX violations that are due to our tracking technique are processed in the hooked page fault handler; the vast majority of page faults are efficiently passed on to the original

handler.

4.2.2 Identity Label Generation

An identity label encapsulates all memory regions (sets of consecutive memory pages) of a process' address space that are executed. Since each executable memory region is self-contained and can be modified independently, DYMO tracks them individually through *image hashes* and *region hashes*.

Image and region hashes are cryptographic hashes (currently we use SHA-1) that represent images (i.e., `.exe` files and DLLs) and executable memory regions, respectively. The primary difference between the two types of hashes is that the former refer to image code segments while the latter correspond to all other executable memory allocations. We make this distinction because of the differences in generating the two types of hashes, as discussed later. A basic identity label is generated by aggregating all image and region hashes into a set. In Section 4.3.2, we discuss an optimization step that allows us to compress the size of identity labels significantly.

Since the label is a set of hashes, the constituent image and region hashes can be individually extracted. As a result, the identity label is independent of the exact layout of executable memory regions in the process' address space (which can change between executions). Furthermore, the identity label encapsulates DLLs that are dynamically loaded according to the run-time behavior of a particular process execution (e.g., the dynamic loading of a JavaScript engine by a browser when rendering a web page that contains JavaScript). The creation of image and region hashes is described next.

Image Hashes. It is easiest to understand the operation of DYMO by walking through the loading and execution of an application. After a process is started and its initial thread is created – but before execution begins – DYMO is notified through the process creation callback. At this point, DYMO constructs a *process profile* to track the process throughout its execution.

Just before the initial thread starts executing, the image loading callback is invoked to notify DYMO that the application's image (the `.exe` file) and the `Ntdll.dll` library have begun loading. DYMO locates the code segment for each of these images in the process' virtual address space and modifies the page protection to remove execute access from the region. DYMO then adds the original protection (`PAGE_EXECUTE_READ`), the new protection (`PAGE_READONLY`), and the image base address to the process profile.

`Ntdll.dll` is responsible for loading all other required DLL images into the process, so the initial thread is set to execute an initialization routine in `Ntdll.dll`. Note that this marks the first user-mode execution attempt in the new process. Since DYMO has removed execute access from the `Ntdll.dll` code segment, the execution attempt raises a DEP/NX exception, which results in a control transfer to the page fault handler. DYMO's page fault handler hook is invoked first, which allows it to inspect the fault. DYMO determines that this is the DEP/NX violation that it induced, and it uses the process profile to match the faulting address to the `Ntdll.dll` code segment. Using the memory region information in the process profile, DYMO creates the image hash that identifies `Ntdll.dll`. It does this by computing a cryptographic hash of the code segment.

Note that special care must be taken to ensure that the image hash is not affected by image rebasing. DYMO accomplishes this by parsing the PE header and `.reloc` section of the image file to find the rebase fixup points and revert them to their canonical values. That is, those addresses in a library's code that change depending on the library's base address are overwritten with their initial values, which are derived from the preferred base address. This is necessary to avoid the generation of different hashes when the same library is loaded at different addresses in different program executions.

The image hash is then added to the process profile. Finally, DYMO restores the original page protection (`PAGE_EXECUTE_READ`) to the faulting region and dismisses the page fault, which allows execution to continue in the `Ntdll.dll` initialization routine.

`Ntdll.dll` consults the executable's Import Address Table (IAT) to find required DLLs to load (and recursively consults these DLLs for imports) and maps them into memory. DYMO is notified of these image loads through a callback, and it carries out the processing described above for each library. The callback is also invoked when DLLs are dynamically loaded during run-time, which enables DYMO to process them as well. After loading, each DLL will attempt to execute its entry point, a DEP/NX exception will be raised, and DYMO will add an image hash for each DLL to the process profile as described above.

Region Hashes. Collecting image hashes allows DYMO to precisely track all of a process' loaded images. But there are other ways to introduce executable code into the address space of a process, such as creating a private memory region or file mapping. Furthermore, the page protection of any existing memory region may be modified to

allow write and/or execute access.

All of these methods eventually translate to requests to one of three system services that are used for memory management, which are hooked by DYMO (namely, `NtAllocateVirtualMemory`, `NtMapViewOfSection`, and `NtProtectVirtualMemory`). When a request to one of these system services is made, DYMO first passes it to the original routine, and then our system checks whether the request resulted in execute access being granted to the specified memory region. If so, DYMO reacts as it did when handling loaded DLLs: it removes execute access from the page protection of the region, and it adds the requested protection, the granted protection, and the region base address to the process profile. When the subsequent DEP/NX exception is raised (when code in the region is executed for the first time), DYMO creates a region hash for the region. Unfortunately, generating a region hash is not as straightforward as creating an image hash (i.e., calculating a cryptographic hash over the memory region). This is because these executable regions are typically used for dynamic code generation, and so the region contents vary wildly over the course of the process' execution. Handling this problem requires additional tracking, which we describe next.

Handling Dynamic Code Generation. To motivate the problem created by dynamic code generation, consider the operation of the Firefox web browser. As of version 3.5, Firefox uses a component called TraceMonkey [113] as part of its JavaScript engine to JIT compile *traces* (hot paths of JavaScript code), and it executes these traces in an allocated memory region. Since the generated code will vary depending upon many factors, it is difficult to track and identify the region (a similar issue arises with recent

versions of Adobe's Flash player and other JIT compiled code). Nonetheless, care must be taken to effectively track the JIT code region as it represents a writable and executable memory region that may be the target of JIT spraying attacks [16].

To overcome this difficulty, DYMO tracks the images that are responsible for allocating, writing, and calling into the region in question. The allocator is tracked by traversing the user-mode stack trace when the region is allocated until the address of the code that requested the allocation (typically a call to `VirtualAlloc`) is reached. DYMO tracks the writer by filtering write access from the region, and, in the page fault handler, capturing the address of the instruction that attempts the write. The caller is tracked by locating the return address from the call into the region. In the page fault handler, this return address can be found by following the user-mode stack pointer, which is saved on the kernel stack as part of the interrupt frame. DYMO creates a (meta) region hash by concatenating the image hashes of the allocator, writer, and caller of the region and hashing the result. In the case of Firefox TraceMonkey, a hash that describes that the region belongs to its JavaScript engine housed in `Js3250.dll` is generated.

Dynamic code rewriting is handled in a similar fashion. Code rewriting occurs, for example, in the Internet Explorer 8 web browser when `Ieframe.dll` rewrites portions of `User32.dll` to detour [72] functions to its dynamically generated code region. In this case, since `User32.dll` has already been registered with the system and DYMO is able to track that `Ieframe.dll` has written to it, the `User32.dll` image hash is updated to reflect its trusted modification.

Handling the `PAGE_EXECUTE_READWRITE` Protection. When a process makes a call that results in a memory protection request that includes both execute and write access, DYMO must take special action. This is because DYMO must allow both accesses to remain transparent to the application. However, it must also differentiate between the two, so that it can reliably create hashes that encapsulate any changes to the region. The solution is to divide the `PAGE_EXECUTE_READWRITE` protection into `PAGE_READWRITE` and `PAGE_EXECUTE_READ` and toggle between the two.

To this end, DYMO filters the `PAGE_EXECUTE_READWRITE` request in a system service hook and, initially, only grants `PAGE_READWRITE` to the allocated region. Later, if the application attempts to execute code in the region, a DEP/NX exception is raised, and DYMO creates a hash as usual, but instead of granting the originally requested access, it grants `PAGE_EXECUTE_READ`. In other words, DYMO removes the write permission from the region so that the application cannot update the code without forcing a recomputation of the hash.

If a fault is later incurred when writing to the region, DYMO simply toggles the protection back to `PAGE_READWRITE` and dismisses the page fault. This strategy allows DYMO to compute a new hash on every execution attempt, while tracking all writes and remaining transparent to the application.

4.2.3 Establishing Identity

Recall that a label is a set of hashes (one for each executable memory region). One way to establish identity is to associate a specific label with an application. A process

is identified as this application only when their labels are identical; that is, for each hash value in the process' label, there is a corresponding hash in the application's label. We call this the *strict matching* policy.

A limitation of the strict matching policy is that it can be overly conservative, rejecting valid labels of legitimate applications. One reason is that an application might not always load the exact same set of dynamic libraries. This can happen when a certain application feature has not been used yet, and, as a result, the code necessary for this feature has not been loaded. For example, consider the case of dynamic code generation in a web browser. When the user has not yet visited a web page that triggers this feature, the label will not contain an entry for a dynamically allocated, executable region created by the JIT compiler. To address this issue, we propose a *relaxed matching* policy that accepts a process label as belonging to a certain application when this process label contains a subset of the hashes that the application label contains *and* the hash for the main code section of the application is present.

4.3 Applications for DYMO

DYMO implements a dynamic code identity primitive. This primitive has a variety of applications, both on the local host and in the network. In this section, we first describe a scenario where DYMO is used for performing local (host-based) access control using the identity of processes. Then, we present an application where DYMO is extended to label network connections based on the program code that is the source of the traffic.

4.3.1 Application-Based Access Control

Modern operating systems typically make access control decisions based on the user ID under which a process runs. This means that a process generally has the same access rights as the logged-in user. DYMO can be used by the local host to enable the OS to make more precise access control decisions based on the identity of applications. For example, the OS could have a policy that limits network access to a set of trusted (whitelisted) applications, such as trusted web browsers and email clients. Another policy could impose restrictions on which applications are allowed to access a particular sensitive file (similar to sealed storage [39]). Because DYMO precisely tracks the dynamic identity of a process, a trusted (but vulnerable) application cannot be exploited to subvert an access control policy. In particular, when a trusted process is exploited, its identity label changes, and, thus, its permissions are implicitly taken away.

To use application-based access control, a mechanism must be in place to distribute identity labels for trusted applications, in addition to a set of permissions that are associated with these applications. The most straightforward approach for this would be to provide a global repository of labels so that all hosts that run DYMO could obtain identity labels for the same applications. We note that global distribution mechanisms already exist (such as Microsoft Update), which DYMO could take advantage of. This would work well for trusted applications that ship with Windows, and they could be equipped with default privileges.

Furthermore, it is also straightforward for an administrator to produce a whitelist of identity labels for applications that users are allowed to run, for example, in an enter-

prise network. To this end, one simply needs to run an application on a system where DYMO is installed, exercising the main functionalities so that all dynamic libraries are loaded. The identity label that our system computes for this application can then be readily used and distributed to all machines in the network. In this scenario, an administrator can restrict the execution of applications to only those that have their labels in a whitelist, or specific permissions can be enabled on a per-application basis.

One may argue that during this training period, it may not be feasible to fully exercise an application so as to guarantee that all possible dynamic libraries are loaded. The problem is that, after DYMO is deployed, untrained paths of execution could lead an application to load unknown libraries that would invalidate the application's identity label, resulting in a false positive. We believe that such problems can be mitigated by focused training that is guided by the users' intended workflow. Furthermore, an administrator may accept a small number of false positives as a trade-off against spending more time to reveal an application's esoteric functionality that is rarely used.

4.3.2 DYMO Network Extension

In this section, we describe our implementation of an extension to DYMO to inject a process' identity label into the network packets that it sends. This allows network entities to learn the provenance of the traffic. An example scenario that could benefit from such information is an enterprise deployment.

In a homogeneous enterprise network, most machines will run the same operating system with identical patch levels. Moreover, a centralized authority can enforce the soft-

ware packages that are permissible on users' machines. In this scenario, it is easy to obtain the labels for those applications and corresponding libraries that are allowed to connect to the outside Internet (e.g., simply by running these applications under DYMO and recording the labels that are observed). These labels then serve as a whitelist, and they can be deployed at the network egress points (e.g., the gateway). Whenever traffic with an invalid label is detected, the connection is terminated, and the source host can be inspected.

By analyzing labels in the network, policies can be enforced at the gateway, instead of at each individual host, which makes policy management simpler and more efficient. Furthermore, the DYMO network extension allows for other traffic monitoring possibilities, such as rate limiting packets from certain applications or gathering statistics pertaining to the applications that are responsible for sending traffic through the network.

To demonstrate how identity labels can be used in the network, we implemented the DYMO network extension as a kernel module that intercepts outbound network traffic to inject all packets with the identity label of the originating process. We accomplish this by injecting a custom IP option into the IP header of each packet, which makes it easy for network devices or hosts along the path to analyze the label. In addition, as an optimization, the label is only injected into the first packet(s) of a TCP connection (i.e., the SYN packet).

The *injector*, a component that is positioned between the TCP/IP transport driver and the network adapter, does the injection to ensure that all traffic is labeled. A second

component, called the *broker*, obtains the appropriate identity label for the injector. These components are discussed next.

The Injector

The injector component is implemented as a Network Driver Interface Specification (NDIS) Intermediate Filter driver. It sits between the TCP/IP Transport Provider (`Tcp-ip.sys`) and the network adapter, which allows it to intercept all IP network traffic leaving the host. Due to the NDIS architecture, the injection component executes in an arbitrary thread context. Practically speaking, this means that the injector cannot reliably determine on its own which process is responsible for a particular network packet. To solve this problem, the injector enlists the help of a broker component (discussed below).

When the injector receives a packet, it inspects the packet headers and builds a *connection ID* consisting of the source and destination IP addresses, the source and destination ports, and the protocol. The injector queries the broker with the connection ID and receives back a process identity label. The label is injected into the outbound packet as a custom IP option, the appropriate IP headers are updated (e.g., header length and checksum), and the packet is forwarded down to the network adapter for delivery.

The Broker

The broker component assists the injector in obtaining appropriate identity labels. The broker receives a connection ID from the injector and maps it to the ascribed process. It then obtains the label that is associated with the given process and returns it to the injector.

The broker is implemented as a Transport Driver Interface (TDI) Filter driver. It resides above `Tcpip.sys` in the transport protocol stack and filters the TDI interfaces that are used to send packets. Through these interfaces, the broker is notified when a process sends network traffic, and it parses the request for its connection ID. Since the broker executes in the context of the process that sends the network traffic, it can maintain a table that maps connection IDs to the corresponding processes.

Label Size Optimization

Identity labels, which store all image and region hashes for a process, can become large. In fact, they might grow too large to fit into the IP option field of one, or a few, network packets. For example, consider the execution of Firefox. It is represented by 87 image and region hashes, each of which is a 20 byte hash value, which results in an identity label size of 1.74 KB. To compress identity labels before embedding them into network packets, DYMO uses Huffman encoding to condense image and region hashes into image and region codes. DYMO then simply concatenates the resulting image and region codes to generate the label that is sent over the network.

The Huffman codes are precomputed from a global input set which includes all trusted applications and DLLs (with their different versions), with shorter codes being assigned to more popular (more frequently executed) images. The codes are stored in a lookup table when DYMO begins operation. To generate a Huffman code for an image hash, the system uses the computed hash of the image to index into the lookup table and obtain the corresponding Huffman code. If the lookup fails, DYMO generates an `UNKNOWN IMAGE` code to describe the image; thus, untrusted or malicious images are easily detected. To generate a region code, DYMO uses the hashes of the allocator, writer, and caller of the region to compute a hash to index into the lookup table. If the lookup fails, DYMO generates an `UNKNOWN REGION` code to describe the region.

In the current implementation, Huffman codes vary in length from 6 to 16 bits. When using optimized codes, DYMO generates an identity label for Firefox that is 74 bytes, which is 4.25% of its size in the unoptimized case. Note that the maximum size of the IP option is fixed at 40 bytes. For identity labels that exceed this 40 byte limit, we split the label over multiple packets.

4.4 Evaluation

We evaluated DYMO on three criteria that address the system requirements that we discussed in Section 4.1.1: the precision of the identity labels it creates, its ability to correctly reflect changes to the identity label when a process has been tampered with, and its impact on application performance.

4.4.1 Label Precision

In order for an identity label to be meaningful, it must uniquely identify the running application that it represents. That is to say, two different applications should receive different labels, and the same application should receive the same label when it is executed multiple times on the same or different hosts. We say that a label meeting these criteria is *precise*.

To evaluate the precision of DYMO's identity labels, we deployed the Windows XP SP3 operating system on three different platforms: a virtual machine running under VMware Fusion 2 on a Mac OS X host, a virtual machine running under VirtualBox 3.1 on an Ubuntu host, and a standard, native installation on bare metal. We then created a test application suite of 107 executables taken from the Windows `System32` directory. To conduct the experiment, we first obtained our database of identity labels using the training method described in Section 4.3.1, that is, by simply running the applications on the test platforms and storing the resulting labels. We then ran each application from the test suite on every platform for ten seconds and for three iterations. In addition, we performed similar tests for Internet Explorer, Firefox, and Thunderbird, which are examples of large and complex applications. For these programs, instead of only running the applications for ten seconds, we simulated a typical workflow that involved browsing through a set of websites – including sites containing JavaScript and Flash content – with Internet Explorer and Firefox and performing mail tasks in Thunderbird.

We found that in all cases, the generated identity labels were precise. There were small differences in the dynamic loading of a few DLLs in some of the processes,

but according to the relaxed matching policy for establishing identity as described in Section 4.2.3, all processes were accepted as belonging to their corresponding applications. More specifically, for 99 of the 107 programs (93%), as well as for Firefox and Thunderbird, the generated labels were identical on all three platforms. In all other cases, the labels were identical among the three runs, but sometimes differed between the different platforms. The reason for the minor differences among the labels was that a particular library was not present (or not loaded) on all platforms. As a result, the applications loaded a different number of libraries, which led to different labels. For six programs, the problem was that the native host was missing an audio driver, and our test suite contained several audio-related programs such as `Mplay32.exe`, `Sndrec32.exe`, and `Sndvol32.exe`. In one case, the VirtualBox platform was missing DLLs for AppleTalk support. In the final two cases (`Magnify.exe` and Internet Explorer), the VirtualBox environment did not load `Msvcp60.dll`.

Our experiments demonstrate that identity labels are precise across platforms according to the relaxed matching policy. In some special cases, certain libraries are not present, but their absence does not change the fundamental identity of the application.

4.4.2 Effect of Process Tampering

An identity label encodes the execution history of a process. We can leverage this property for detecting suspicious behavior of otherwise benign processes when they are tampered with by malware or exploits.

Tampering by Malware

We identified three malware samples that perform injection of code into the address space of other running processes. The first sample was a Zeus bot that modified a running instance of Internet Explorer by injecting code into `Browseui.dll` and `Ws2-help.dll`. The second sample was a Korgo worm that injected a remote thread into Windows Explorer and loaded 19 DLLs for scanning activity and communication with a Command and Control (C&C) server. The third sample was a suspicious program called YGB Hack Time that was detected by 33 out of 42 (79%) antivirus engines in VirusTotal. [2] YGB injected a DLL called `Itrack.dll` into most running processes, including Internet Explorer.

We executed the three samples on a virtual machine with DYMO running. The identity labels of the target applications changed after all three malware samples were executed and performed their injection. This demonstrates that DYMO is able to dynamically update a process' identity label according to changes in its execution.

Tampering by Exploits

An alternative way to tamper with a process' execution is through an exploit that targets a vulnerability in the process. Two common attack vectors are the buffer overflow exploit and drive-by download attack. To demonstrate DYMO's ability to detect such attacks, we used the Metasploit Framework [3] to deploy a VNC server that targets a buffer

[2] `http://www.virustotal.com`
[3] `http://www.metasploit.com`

overflow vulnerability in RealVNC Client and a web server to simulate the Operation Aurora drive-by download exploit [218]. For both attacks, we configured Metasploit to use a sophisticated Reflective DLL Injection exploit payload [43] that allows a DLL to load itself into the target address space without using the facilities of the `Ntdll.dll` image loader. This makes the injection stealthier because the DLL is not registered with the hosting process (e.g., the DLL does not appear in the list of loaded modules in the Process Environment Block).

We deployed our attack VNC server and web server and navigated to them using a vulnerable version of RealVNC Client and Internet Explorer, respectively. The identity labels changed for both vulnerable applications after the attack because of the execution of code in RealVNC Client's stack, Internet Explorer's heap, and the DLL injected into the address space of both. This demonstrates that DYMO is able to update a process' identity label even in the face of a sophisticated attack technique designed to hide its presence.

4.4.3 Performance Impact

DYMO operates at a low level in the Windows XP kernel and must track when a process loads DLLs and makes memory allocation or protection change requests. Moreover, the system adds logic to the page fault handler. Since these kernel functions are frequently invoked, care must be taken to maintain an acceptable level of performance.

Typically, a process will perform most, if not all, of the code loading work very early in its lifetime. Figure 4.1 shows an example of DLL loading over time for Internet Ex-

plorer, Firefox, and Thunderbird (only load-time DLLs are included). Note that 95%, 93%, and 97% of the DLLs were loaded within one second after launching Internet Explorer, Firefox, and Thunderbird, respectively.

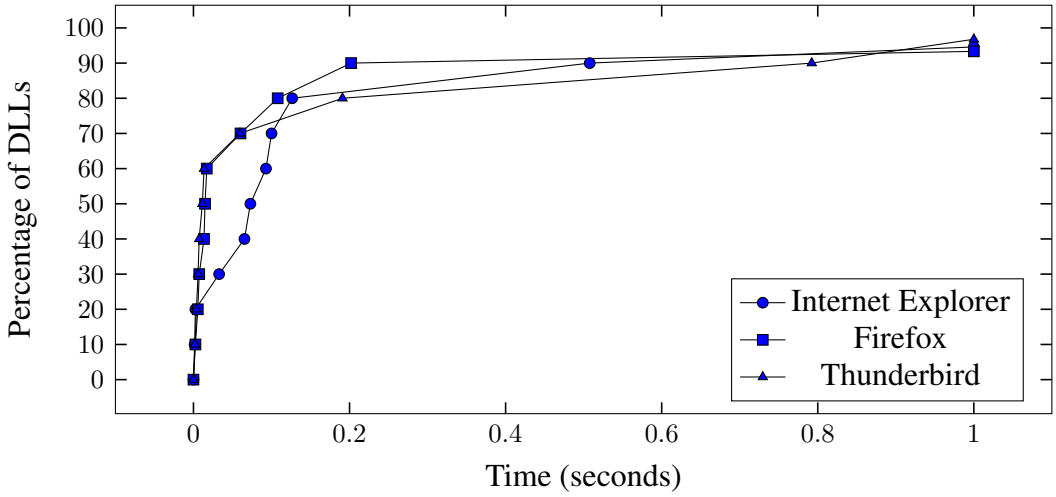


Figure 4.1: DLL loading over time.

The loading of DLLs results in the most work (and overhead) for DYMO, because it means that the system has to compute hashes for new code pages. Thus, the overhead during startup constitutes a worst case. To measure the startup overhead, we ran Internet Explorer, Firefox, and Thunderbird on the native platform, and we measured the time until each application's main window responded to user input with and without DYMO. We used the PassMark AppTimer [4] tool to do these measurements. Table 4.1 shows the results. It can be seen that, with our system running, the startup times for Internet Explorer, Firefox, and Thunderbird increased by 80%, 41%, and 31%, respectively. While the overhead for Internet Explorer may seem high, note that the browser still starts in less than one second. We feel that this is below the threshold of user

[4] `http://www.passmark.com/products/apptimer.htm`

awareness; therefore, it is an acceptable overhead. We speculate that the higher overhead of Internet Explorer can be attributed to its multi-process, Loosely-Coupled IE (LCIE) architecture [215], which results in DYMO duplicating its initialization efforts over the frame and tab processes.

Table 4.1: Application startup overhead (in milliseconds) with DYMO.

Application	Without DYMO	With DYMO	Overhead
Internet Explorer	447	804	80%
Firefox	450	634	41%
Thunderbird	799	1047	31%

In addition to the worst-case overhead during application startup, we were also interested in understanding the performance penalty due to our modifications to the memory management routines and, in particular, the page fault handler. To this end, we wrote a tool that first allocated a 2 GB buffer in memory and then stepped through this buffer, accessing a byte on each consecutive page. This caused many page faults, and, as a result, it allowed us to measure the overhead that a memory-intensive application might experience once the code regions (binary image and libraries) are loaded and the appropriate identity label is computed. We ran this test for 20 iterations and found that DYMO incurs a modest overhead of 7.09% on average.

4.5 Security Analysis

In this section, we discuss the security of our proposed identity label mechanism. In our threat model, we assume that the attacker controls a malicious process and wants

to carry out a security sensitive operation that is restricted to a set of applications with known, trusted identities. Similarly, the attacker might want to send a network packet with the label of a trusted process.

The malicious process could attempt to obtain one of the trusted labels. To this end, the attacker would have to create executable memory regions that hash to the same values as the memory regions of a trusted process. Because we use a strong hash function (SHA-1), it is infeasible for the attacker to allocate an executable region that hashes to a known value. It is also not possible to simply add code to a trusted program in order to carry out a sensitive operation on the attacker's behalf (a kind of confused deputy attack [67]). The reason is that any added executable region would contribute an additional, unknown hash value to the identity label, thereby invalidating it.

A malware process could also attempt to tamper with the data of a process and indirectly modify its operations so that it could carry out malicious activity. This is a more difficult attack, and its success depends on the normal functionality that is implemented by the targeted victim program. The easiest way to carry out this attack is via a debugger, which allows easy manipulation of the heap or stack areas of the victim application. We prevent this attack by disabling access to the Windows debugging API for all user processes when our system is running. We believe that these APIs are only rarely used by regular users, and it is reasonable to accept the reduced functionality for non-developers.

Another way to tamper with the execution of an application without injecting additional code is via non-control-data attacks. These attacks modify "decision-making data" that

might be used by the application while carrying out its computations and interactions. Previous work [21] has shown that these attacks are "realistic threats," but they are significantly more difficult to perform than attacks in which arbitrary code can be injected. Moreover, for these attacks to be successful, the malware has to find an application vulnerability that can be exploited, and this vulnerability must be suitable to coerce the program to run the functionality that is intended by the malware author. Our current system does not specifically defend against these attacks. However, there are a number of operating system improvements that make exploits such as these significantly more difficult to launch. For example, address space layout randomization (ASLR) [14] provides a strong defense against attacks that leverage return-oriented programming (advanced return-into-libc exploits) [171]. Because our technique is compatible with ASLR, our system directly benefits from it and will likely also profit from other OS defenses. This makes this class of attacks less of a concern.

4.6 Summary

We presented DYMO, a system that provides a dynamic code identity primitive that enables tracking of the run-time integrity of a process. Our system deploys a host-based monitoring component to ensure that all code that is associated with the execution of an application is reliably tracked. By dynamically monitoring the identity of a process in a trustworthy fashion, DYMO enables an operating system to enforce precise application-based access control policies, such as malware detection, application whitelisting, and providing different levels of service to different applications. In addition, we imple-

mented an application that extends DYMO so that network packets are labeled with information that allows one to determine which program is responsible for the generation of the traffic. We have developed a prototype of our approach for the Windows XP operating system, and we have evaluated it in a number of realistic settings. The results show that our system is able to reliably track the identity of an application while incurring an acceptable performance overhead.

Chapter 5

Containing Malware by Tracking Process Interactions

In the previous chapter, we described how DYMO can be used to effectively detect sophisticated malware. For example, recall that the Torpig bot (discussed in Chapter 3) is implemented as a set of DLLs that are injected into a variety of trusted applications, such as web browsers and email clients. DYMO captures the changes that are made to these applications because modules that contain code are injected by the malware, thereby altering each target process' identity. However, there are ways in which malware can manipulate trusted processes without adding any executable code (these mechanisms are discussed in Section 5.2). Malware that uses such techniques may be able to evade the detection capability that DYMO provides, since a process' identity changes only when its code is modified. Incidentally, these attacks are possible only because of certain design choices for modern operating systems.

To wit, contemporary operating systems are designed to offer a flexible platform that supports a diverse set of applications. As such, performance and ease of software development are primary design considerations. These design goals influence an important system trade-off between two diametrically opposed features: process isolation and inter-process communication. Process isolation ensures that a process cannot corrupt the data or code of another process. Inter-process communication provides a channel over which data, code, and commands can be exchanged. If a system isolates processes weakly, then communication is straightforward, for example, by allowing processes to directly read from and write to each other's memory. In contrast, strongly isolated processes may only communicate in ways that are more complicated and performance-intensive, for example, by using remote procedure calls (RPCs).

Completely isolated processes are secure against external tampering, but they are not very useful in and of themselves. Inter-process communication features provide a mechanism by which processes can share data and, perhaps, influence each other's execution in a controlled manner. Thus, process isolation suffers as inter-process communication becomes more flexible and easy to use. Due to the design goals mentioned above, modern systems tend to favor diverse and flexible inter-process communication mechanisms over strong process isolation. Unfortunately, an overly permissive security model combines with weak process isolation to provide an environment in which malware can abuse its ability to freely interact with other processes.

For instance, consider a personal firewall that limits egress network traffic to a few trusted applications, such as Internet Explorer for web traffic and Outlook for email. If a malicious process directly attempts to make an Internet connection, the firewall can

successfully block the attempt, as it represents a violation of the policy. However, the malicious process can send inputs to the Internet Explorer application in order to drive the browser to execute the request on behalf of the malware. For example, the malware might utilize a COM interface that Internet Explorer exposes or, more generically, it may directly write code into the browser's address space and execute it. In this case, the firewall sees a network request coming from the trusted Internet Explorer process, so it allows the traffic to pass through unabated. This is an example of the *confused deputy problem* [67], and it is characteristic of the access control systems that contemporary operating systems use.

As another example, consider a keylogger that intercepts a user's sensitive keyboard inputs (e.g., online banking account credentials). This can be accomplished by attaching the input processing of a thread in the keylogger to that of a thread in a trusted process that is responsible for handling sensitive input (e.g., a web browser). It is difficult for a malware detection system to attribute this attack to the keylogger, because input messages that are destined for the trusted process will be sent to both threads.

These attacks demonstrate that weak process isolation is problematic for host-based detection tools that need to attribute events to processes. This is because, without strong isolation, systems are vulnerable to abuse by malware that interacts with another (benign) process to carry out malicious actions on its behalf. In this chapter, we propose PRISON, a system that monitors process interactions to detect and mitigate malicious inter-process communication and code or data injection. [1] In particular, we developed a host-based component that monitors the system calls that enable process interactions.

[1] An earlier version of this work was in submission at the time of this writing.

PRISON can detect when a system call is used by one process to interact with another, and the system can then automatically block the communication if it determines that the interaction is malicious.

Tracking process interactions in a comprehensive manner is difficult, because operating systems provide a diverse set of communication interfaces over protocols that are often undocumented. As evidence of this difficulty, we evaluated a number of host-based security products, and we show that these systems fail to capture all process interactions (these experiments are described in Section 5.4.2). This failure implies that malware is able to effectively "launder" its malicious actions through trusted processes to avoid detection and containment.

We implemented our system as a Windows XP kernel extension. PRISON hooks a variety of kernel system services (system calls) in order to monitor a diverse set of interaction channels. Our experiments demonstrate that PRISON is effective at blocking unauthorized process interactions over all such mechanisms that we are aware of. Furthermore, we show that our efficient technique offers finer granularity and better coverage than existing commercial tools.

5.1 System Overview

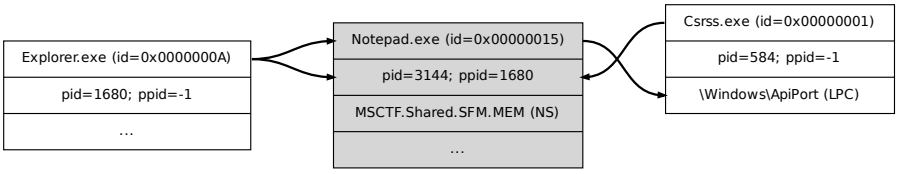
Our solution to address the problem of malicious process interaction is PRISON, a host-based system that operates as a kernel extension to the Windows XP platform. In order to get a detailed understanding of how inter-process communication is implemented in

Windows, we performed a deep analysis of the communication mechanisms that are available in the system (the analysis is described in Section 5.2). PRISON monitors all known kernel interfaces to detect when processes interact. The system then blocks the communication attempt if it determines that the source violates a security policy.

Modes of Operation. PRISON has two modes of operation: *logging mode* and *filtering mode*. In logging mode, PRISON maintains a real-time *interaction log* of all interactions between processes. The system also generates an *interaction graph* that depicts these interactions. For each process, the log includes an entry that enumerates the interactions that involve the process as a target (e.g., an RPC client request or a write into the process' memory space). The log also contains related information, such as the list of LPC ports or named memory sections that the process has opened. The interaction graph offers a visual representation that concisely depicts a set of interactions. Because viewing an interaction graph of the whole system quickly becomes unmanageable, PRISON can focus the graph on specific processes that are of interest. Figure 5.1 shows an example entry from the interaction log and a corresponding interaction graph for the launch of `Notepad.exe`. The figure shows Notepad interacting with two processes: the Windows shell (`Explorer.exe`) and the Windows subsystem process (`Csrss.exe`). The former is the parent of Notepad, which accounts for the process and thread creation (represented in Figure 5.1b by the arrow pointing to the first element of the `Notepad.exe` node) and memory reading and writing events (depicted by the arrow pointing to the second `Notepad.exe` element). `Csrss.exe` shares a memory region with Notepad (and all Windows processes) as part of its initialization (e.g., to set up its debugging and exception ports).

```
Process Notepad.exe (uid=0x00000015;pid=3144;ppid=1680) ended:0
  List of ports owned by this process:
    Port MSCTF.Shared.SFM.MEM (type=Named section; subtype=None)
    ...
  List of processes that have interacted with this target:
    Process Explorer.exe (uid=0x0000000A;pid=1680;ppid=-1)
      Process creation (NtCreateProcessEx) port=-; blocked=False; nb=1
      Reading memory (NtReadVirtualMemory) port=-; blocked=False; nb=11
      Allocating memory (NtAllocateVirtualMemory) port=-; blocked=False; nb=5
      Writing memory (NtWriteVirtualMemory) port=-; blocked=False; nb=5
      Executing code (NtCreateThread) port=-; blocked=False; nb=1
    Process Csrss.exe (uid=0x00000001;pid=584;ppid=-1)
      Injecting data (NtMapViewOfSection) port=-; blocked=False; nb=5
      Writing memory (NtWriteVirtualMemory) port=-; blocked=False; nb=4
      Reading memory (NtReadVirtualMemory) port=-; blocked=False; nb=4
    ...
```

(a) interaction log entry



(b) interaction graph

Figure 5.1: Sample interaction log entry and corresponding interaction graph.

PRISON can be queried as a service by other malware detection systems that may use the interaction information for their real-time analysis. Our system makes this feature available through an input and output control (IOCTL) interface. Furthermore, PRISON can be used as a dynamic malware analysis tool. To this end, an analyst can configure the system to write its interaction log and graph to disk so that they can be used for offline analysis. The interaction log could also be included in the reports of malware analysis sandboxes, such as Anubis [75] or CWSandbox [208].

In filtering mode, PRISON interposes on each process interaction attempt, and it makes a policy-based decision that determines whether or not this interaction is allowed. If the interaction is permitted under the policy, then the communication request passes

unabated to the system service that is responsible for fulfilling the request. However, if the interaction attempt indicates a policy violation, then it is blocked. In this case, the source process is returned an error code that indicates that the target is unavailable and the communication fails.

Interaction Filtering Policies. We have identified two *interaction filtering policies* for use in PRISON's filtering mode. The first policy is a whitelisting approach in which a list of system processes (e.g., `Explorer.exe` and `Csrss.exe`) and trusted applications are granted access to the inter-process communication facilities that they require, and all other interactions are blocked. A straightforward way to identify applications for such a whitelist is by process name (including the path), similar to how the "path rule" operates in the Windows Software Restriction Policies mechanism [129]. However, if a stronger notion of identity is required, we could leverage existing work in the area of code identity, including the use of a TPM [166] or a dynamic code identity primitive [58].

An alternative filtering policy is a *process tainting* approach. This policy requires an additional kernel module that augments the Windows access control mechanism with a set of high-level system access rights, such as *file system*, *registry*, and *network*. These rights are granted to a process according to its needs when it is created, and the rights are enforced when the process attempts the corresponding access. Whenever a source communicates with a target, the latter obtains the *intersection* of the two process' respective high-level rights. For example, assume that the Notepad text editor and Internet Explorer browser are granted {*file system*, *registry*} and {*file system*,

registry, *network*} high-level access rights, respectively. If Notepad attempts to communicate with Internet Explorer (perhaps due to an attack against the text editor), then the browser is left with {*file system*, *registry*} rights, and, thus, would be unable to use the network, since it has been "tainted" by Notepad.

The policy-based filtering mode makes PRISON a powerful tool for detecting and mitigating the threat of malicious process interactions. This is important because malware authors have a variety of options at their disposal for leveraging a benign process to perform actions on their behalf. We describe the design and implementation of PRISON in detail in Section 5.3, but, before doing so, we first discuss our analysis of the techniques for process interaction that are available in Windows.

5.2 Process Interaction Mechanisms

The mechanisms that allow one process to interact with another can be broadly classified into two categories: (1) standard inter-process communication interfaces, which are specifically implemented by particular applications (e.g., COM) and (2) techniques that make use of platform-provided system services that enable the injection of code or data into the address space of the target application (e.g., remote thread injection). The work that we are presenting in this chapter focuses on detecting malicious process interaction within the Windows operating system; therefore, we limit our discussion to the mechanisms that are available on that platform (and on Windows XP specifically). Nevertheless, other operating systems offer similar functionality to allow processes to communicate.

5.2.1 Inter-process Communication Techniques

Well-established communication standards allow applications to provide useful services to other programs, which encourages componentization and obviates the need to duplicate functionality. This inter-process communication is accomplished by having applications support specific mechanisms that use various facilities that are provided by the operating system. The standardized communication interfaces that Windows makes available are described in this section.

Windows Messages. Windows graphical applications are event-driven programs that revolve around a message loop. Events are sent to an application as Windows messages, which are generated at each input event (e.g., a keyboard press or mouse click). Events are generated by the operating system itself or by applications, such as when an application's window is resized. Furthermore, one process can send synthetic messages to another process. For example, the popular AutoIt automation language makes extensive use of Windows messages to drive applications.

Dynamic Data Exchange. Dynamic Data Exchange (DDE) is a legacy inter-process communication mechanism that was introduced in Windows 2.0. DDE is fundamentally a message-passing protocol and is built upon Windows messages. It was primarily designed to allow processes to share data, but it is also commonly used to control other applications. For example, Internet Explorer can be used to navigate to a specific URL by establishing a DDE *conversation* with the `WWW_OpenURL` command.

Shared Memory. Windows offers a shared memory facility to provide an efficient mechanism for two processes to exchange data. This functionality is implemented as a special case of the file mapping APIs, whereby the created file mapping object is backed by the page file instead of an actual file on disk. Processes may opt to use shared memory because of its performance benefits compared to other communication mechanisms. However, the communicating endpoints are required to manage issues such as message notification and synchronization.

Named Pipe and Mailslot Communication. A named pipe on the Windows platform is an abstraction of shared memory that processes can use for communication. One process, called the *pipe server*, creates the pipe and assigns it a name. Subsequently, a *pipe client* can connect to the pipe to send and receive data. The pipe server has the responsibility of creating a protocol that its clients can use to request services or send data. Mailslots operate almost identically to named pipes. They are designed for one-way communication, and they support broadcast within a network domain. The interface to named pipes and mailslots is semantically equivalent.

Local Procedure Call. Local Procedure Call (LPC) is a communication paradigm that supports high-speed message passing. LPCs are not intended to be directly used by user applications but rather by operating system components on their behalf. As such, the LPC APIs are native API functions that are exported by `Ntdll.dll` in user-mode. LPCs use a client-server model, whereby the server first registers a named LPC connection port object to which clients connect. Once a connection is made with a particular client, the client and server each create a communication port over which

small messages are exchanged and the scheme for passing large messages is negotiated (e.g., using shared memory).

Remote Procedure Call. Remote Procedure Call (RPC) is a network programming model for local or remote inter-process communication that builds upon other APIs, such as Winsock (i.e., TCP/IP), named pipes, or LPC. RPC provides an endpoint-agnostic, procedural view of network operations, which simplifies distributed application development, as the location of the target (local or remote) is abstracted away by the model. RPC is a commonly used method for inter-process communication, and it forms the basis for the Component Object Model.

Component Object Model. The Component Object Model (COM) is a standard for creating binary software components that can interact. Inter-object communication is accomplished in COM through the use of RPC. COM allows a process to dynamically access an interface that is exposed by a component and use the methods of that interface to communicate with the component. For example, Internet Explorer exports the `IWebBrowser2` COM interface, which exposes a variety of methods. One such method, `Navigate`, can be used to have the browser visit a URL that is specified by the client.

5.2.2 Injection Techniques

Most applications that intend to exchange data or provide functionality to other processes do so through the documented interfaces that were described in the previous section. However, these inter-process communication techniques are not the only way for processes to interact. In particular, a (malicious) process can use a number of system calls to directly inject code or data into the address space of another process (e.g., `WriteProcessMemory`). Windows also offers mechanisms that allow a process to indirectly coerce another process to load and execute arbitrary code. In both cases, the result is that the target process will perform tasks on behalf of the (possibly malicious) instigating process. These techniques are discussed in this section.

Direct Injection

There are three primary steps that a malicious process must take to directly force another process to execute injected code. First, the malware needs to obtain a valid handle to the target process. Second, the malware must perform the injection of code into the target's address space. Finally, the target must be induced to execute the injected code.

It is trivial for a malicious process to obtain a handle to its target process. For example, the `OpenProcess` API returns a handle to a specified process. One caveat is that the caller must request a set of access rights with respect to the process object that is opened. For the purposes of injection, these access rights must include a request to write into the target process. The requested access rights are subject to the Security Reference Monitor's access check, in which the caller's access token, the target's secu-

rity descriptor, and the requested access rights are used to allow or deny the access. By default, processes are created with a security descriptor that grants write access to other processes that run under the same user account. Thus, the access check is typically not an obstacle to injection.

Armed with an appropriate handle to the target process, malware can then inject code into the target in one of two ways. First, code can be directly written into the virtual address space of the target, subject to the page permissions of the memory region. Alternatively, the code can be memory mapped from a file into the target process. While the Win32 API does not allow files to be mapped into other processes, the native NT API does, with a call to `NtMapViewOfSection` (exported by `Ntdll.dll`).

Finally, after the injection is complete, the attacking process must induce the target process to execute the code. Windows supports a variety of mechanisms to accomplish this task. For example, a new thread can be created in the target process and set to begin execution at a specified address through the `CreateRemoteThread` function. Alternatively, the malware can use the debugging API to attach to the target process, handle debug events, and change the context of a thread such that the instruction pointer is set to begin execution at the address of the injected code. Finally, malware can queue a user-mode asynchronous procedure call (APC) to a thread in the target process with the APC function address set to that of the injected code.

Indirect Injection

The Windows operating system supports mechanisms that enable a DLL to be loaded into another process without requiring the attacker to directly inject code or data into its target. We call this *indirect injection*. For these attacks to succeed, the attacker needs only to control malware that has the typical access rights that are granted to a user process.

First, the malware can use the Windows hook message handling feature, which was designed to allow an application to interpose on system events (e.g., keyboard presses or mouse clicks). To use Windows hooks, the caller invokes the `SetWindowsHookEx` function to specify the types of events it is interested in and the hook function to be used as a callback. Hooks can have global scope, meaning that they are installed in all processes in the current desktop, or thread scope, which targets a specific thread. In either case, the caller provides a handle to a DLL and a pointer to the hook function. The DLL is then loaded by the system into the address space of the target (or all processes, if the hook is global), and the hook function is called when the designated event is triggered.

Another indirect injection technique is a legacy feature whereby DLLs that are listed under the `AppInit_DLLs` registry value are loaded into all Windows-based applications (specifically, all applications that link with `User32.dll`) when they are launched. An attacker can leverage this mechanism by placing a DLL on the file system and appending the path to that library to the `AppInit_DLLs` registry value. Since almost all executables are Windows-based applications, the DLL will be loaded into most pro-

cesses.

5.3 System Implementation

In this section, we describe how PRISON is implemented to detect and mitigate malicious process interactions. To interpose on all communication attempts, PRISON must intercept the function calls that implement the interaction techniques. PRISON accomplishes this by hooking the relevant system services in the Windows XP kernel. A few techniques can be used to hook system services [68]. We opted for a standard approach that is used by many security products and other tools that filter system calls (including rootkits); we patch the *System Service Dispatch Table* (SSDT). This table is used by the operating system to dispatch system call requests to the corresponding system services.

5.3.1 Implementing the Interaction Filtering Policy

PRISON utilizes a whitelisting interaction filtering policy in order to allow a set of system processes and trusted applications to interact with other processes. The policy limits these trusted processes to only use the mechanisms that are necessary for them to operate. The whitelist is essentially a file that lists the full path and name of the trusted process and a set of interaction techniques that the process is known to utilize during normal operation. We could automatically generate a whitelist by defining a training period in which we track all normal interactions that occur between trusted processes on the host and then add these interactions to the whitelist. This is similar to

the policy generation mechanisms that exist in other systems, for example, [149]. This filtering policy provides a simple mechanism by which PRISON can prevent malware from interacting with other processes on the host.

5.3.2 Monitoring Interaction-Relevant System Services

In Section 5.2, we discussed an overview of our analysis of the techniques that enable processes to interact on the Windows platform. Fundamentally, each interaction method is implemented by one or more system services. In this section, we describe the services that PRISON must hook to successfully monitor all of these interaction techniques.

Interposing on most of the interaction mechanisms is a straightforward process of identifying and hooking a single corresponding system service. Unfortunately, handling other techniques is more difficult. In particular, some interaction channels require a deep understanding of undocumented structures and interfaces to map relevant objects to the corresponding source and target processes. It is important to capture this mapping, because PRISON requires the identity of the communicating processes for its analysis. In the following, we describe our engineering challenges and our methods of solving them.

Windows Messages. Windows exposes a set of functions that can be used to send Windows messages, including `SendMessage`, `PostMessage`, and `PostThreadMessage`. These functions resolve to calls to a matching set of system services: `Nt-`

`UserMessageCall`, `NtUserPostMessage`, and `NtUserPostThreadMessage`, respectively. To intercept the exchange of Windows messages, PRISON hooks these three system services. For each service, PRISON analyzes the message type parameter to detect the type of interaction (e.g., a `WM_COPY` message indicates a clipboard copy operation). The target process is determined from another parameter that specifies a handle to the window that the message is meant for.

Another way in which malware can abuse Windows messages is to attach to the input processing mechanism of another thread. This technique is often used by keyloggers to capture the keystrokes of the user. A user-mode process can call `AttachThreadInput` to accomplish this task. PRISON hooks the corresponding system service, `NtUserAttachThreadInput`, and identifies the target process as described above.

A particular problem arises when attempting to hook user interface (USER) or Graphics Device Interface (GDI) system services, like the ones described above. While most NT services are handled by the kernel itself, USER and GDI services are implemented by the kernel-mode part of the Windows subsystem (in an extension called `Win32k.sys`). For this reason, USER and GDI functions are exported by a secondary shadow SSDT. This additional SSDT is only mapped into the calling process when one of the corresponding services is first requested. This is problematic in cases where PRISON attempts to hook the USER services in the context of a process that has not mapped the shadow SSDT. PRISON solves this as follows: First, it attaches to a process that is known to have already mapped the shadow SSDT (by calling `KeStackAttachProcess`). Second, PRISON locates the physical address of the table. Third, it reverts to the original process context and uses a memory descriptor list (MDL) to

map the table into that context. Finally, PRISON can safely patch the shadow SSDT.

Dynamic Data Exchange. The Win32 API provides an interface to the Dynamic Data Exchange (DDE) message passing protocol through functions such as `DdeConnect` and `DdeGetData`. Fundamentally, DDE relies on Windows messages to implement the protocol. In particular, a set of messages, delimited by `WM_DDE_FIRST` and `WM_DDE_LAST`, are sent from the source to the target. Thus, monitoring DDE communication reduces to the handling of Windows messages, as we described above.

Shared Memory. Shared memory functionality in Windows is implemented over the file mapping mechanism. In particular, the provider of a shared memory region will first call `CreateFileMapping`, specifying a name for the shared memory region and a special value, `INVALID_HANDLE_VALUE`, in place of the file handle. Next, the provider calls `MapViewOfFile` to actually map the shared memory into the process' virtual address space. Consumers typically request access to the shared memory by calling `OpenFileMapping`.

PRISON hooks both the `NtCreateSection` and `NtOpenSection` system services to analyze creation and access requests to shared memory, respectively. Determining the identity of the target (producer) process is not entirely straightforward since such information is not available when the consumer calls `NtOpenSection`. We devised a novel solution to this problem as follows: First, when the producer invokes `NtCreateSection`, PRISON logs the name of the region and the process ID of the producer in an *identity cache*. Later, when the consumer requests access to the shared

memory, PRISON uses the identity cache to match the name of the region to the producer's ID to identify the target.

Another subtlety is that the consumer may sometimes invoke `CreateFileMapping` (and, thus, `NtCreateSection`) to access the shared memory region. In this case, PRISON distinguishes between create and open operations by intercepting the call and first attempting to open the section itself. If the attempt succeeds, then PRISON interprets the call as an open request by a consumer.

Named Pipe and Mailslot Communication. A server process initializes a named pipe by calling the `CreateNamedPipe` API, providing a unique name to identify the endpoint (e.g., `\\.\Pipe\SomeNamedPipe`). A client subsequently connects to the server by specifying the pipe name in a call to `CreateFile` or `CallNamedPipe`. If successful, the client receives a handle that represents the connection, which it uses to send and receive data over the pipe (by calling `WriteFile` and `ReadFile`, respectively).

It is straightforward to identify the client process of a named pipe communication attempt, because the call is made in the client's context. Determining the identity of the target (i.e., the server) is more challenging, for two reasons. First, the client sends and receives data over the pipe using the generic `WriteFile` and `ReadFile` APIs. These functions take a file handle as a parameter. The Windows Object Manager uses this handle to identify the client side of the pipe connection, but the handle offers PRISON no information that identifies the server. Second, the data that is sent over the pipe represents a custom protocol between the client and server, so no identifying

information can be gleaned from the messages themselves. Our solution is similar to how we handle the identification of the endpoints that communicate over shared memory. In particular, PRISON hooks `NtCreateNamedPipe` to capture the name of the pipe that the server creates along with the process ID of the server. This information is stored in PRISON's identity cache. Later, a client invokes the `NtCreateFile` system service to establish a connection with the server. By hooking this service, PRISON can acquire the pipe name from a parameter and then use the identity cache to match the name to its corresponding server ID.

The operational semantics of named pipe and mailslot communication are nearly identical. A server initializes a mailslot by calling `CreateMailslot`. A client calls `CreateFile` with the name of the mailslot (e.g., `\\.\Mailslot\SomeMailslot`) to get a handle to the mailslot object. The client then uses the mailslot to broadcast messages by invoking `WriteFile`. The mailslot server calls `ReadFile` to read the client's messages. We employ the same strategy to identify mailslot endpoints as we did for named pipes.

Local Procedure Call. The Local Procedure Call (LPC) mechanism is designed to be used only by system processes. However, its API is exposed by `Ntdll.dll`, so it is possible for malware to leverage the protocol for inter-process communication. A server creates an LPC port by invoking the `NtCreatePort` or `NtCreateWaitablePort` system service, supplying a name for the port. A client later connects to the port by calling `NtConnectPort` or `NtSecureConnectPort`. Once a connection is established, both client and server communicate by using handles to their respective

communication port objects.

PRISON analyzes LPC communication by applying the same strategy that is used for named pipes and mailslots. In particular, PRISON stores (in the identity cache) a mapping between the name of the LPC connection port and the corresponding process ID of the server that created it. When a client connects to the port (e.g., by calling `NtSecureConnectPort`), PRISON extracts the port name and matches it to the process ID that it previously stored to identify the server.

Remote Procedure Call. Remote Procedure Call (RPC) is a flexible inter-process communication mechanism that operates over Winsock (i.e., the TCP/IP protocol stack), named pipes, or LPC. Since PRISON is a host-based system, it ignores RPCs that use TCP/IP network communication. The other two cases are handled by the internal mechanisms themselves, as discussed in the corresponding sections above. PRISON is able to distinguish RPC communications that operate over LPCs from other LPCs due to the naming convention that the RPC framework uses when creating its ports. This is beneficial for logging purposes, but it is not strictly necessary for analysis. Unfortunately, no such convention seems to exist for RPCs that operate over named pipes.

Component Object Model. The Component Object Model (COM) is built upon a communication stack that includes RPC operating over the LPC protocol (for local COM components) or TCP/IP (for remote components). Correctly handling communication over COM is particularly challenging for two reasons. The first difficulty is that detecting the target process that hosts one or more COM objects does not offer enough

granularity for PRISON to decide whether or not to block the interaction. The reason is that one process can support multiple COM objects, and we need to distinguish between them. For example, the Background Intelligent Transfer Service (BITS) exposes a COM interface through which a client may transfer files over the Internet. BITS is typically hosted in a shared service process (a Service Host process, `Svchost.exe`) along with a number of other components. In this case, identifying the target process does not allow PRISON to meaningfully analyze the interaction endpoint. Instead, PRISON must identify the particular COM object itself. The second issue is that local COM components dynamically create LPC communication ports with names that do not infer the identity of the endpoint. This makes it exceedingly difficult for PRISON to determine which COM component a particular request is destined for.

We invested a substantial (reverse) engineering effort to distinguish the COM server endpoint in a given communication attempt. This included intercepting requests to the Endpoint Mapper component to obtain the names of dynamic LPC ports and reverse engineering the payloads of various COM-related LPC messages. In particular, we needed to decode custom marshaled object references to obtain the identities of the requested interface and endpoint. With this information, PRISON is able to effectively analyze COM interactions.

Code and Data Injection. As we discussed in Section 5.2.2, there are a number of ways for malware to inject code or data into a target process. These methods result in invocations of the following system services: `NtWriteVirtualMemory`, `NtMapViewOfSection`, `NtCreateRemoteThread`, `NtSetContextThread`, and `Nt-`

`QueueAPCThread`. PRISON hooks all of these services and analyzes them to obtain the source and target processes of the interaction attempt in a straightforward manner.

Windows Hook Injection. A process can install a Windows hook by calling the `SetWindowsHookEx` API. PRISON interposes on the corresponding system service, `NtUserSetWindowsHookEx`. The service takes a thread ID as a parameter, which designates the thread that the hook targets. PRISON uses the thread ID to determine which process the hook is meant to target. However, it is possible for the source to pass a value of zero as the thread ID when invoking the system service. In this case, the hook is to be installed in all threads that are running within the active desktop. PRISON handles this situation by simply blocking the hook request.

5.4 Evaluation

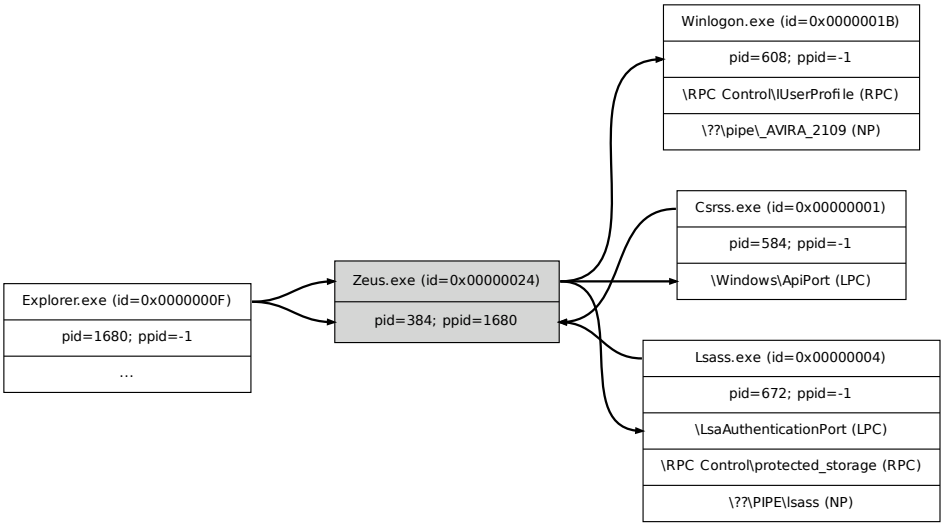
In this section, we discuss the results of our evaluation of PRISON in four areas. First, we show that PRISON can be used to effectively contain malware that attempts to interact with trusted processes. Second, we describe how our system compares to commercial host-based security products. Third, we show that our system comprehensively monitors interactions that occur between processes on the host. Finally, we demonstrate that PRISON introduces only a negligible performance overhead.

5.4.1 Containing Malicious Process Interactions

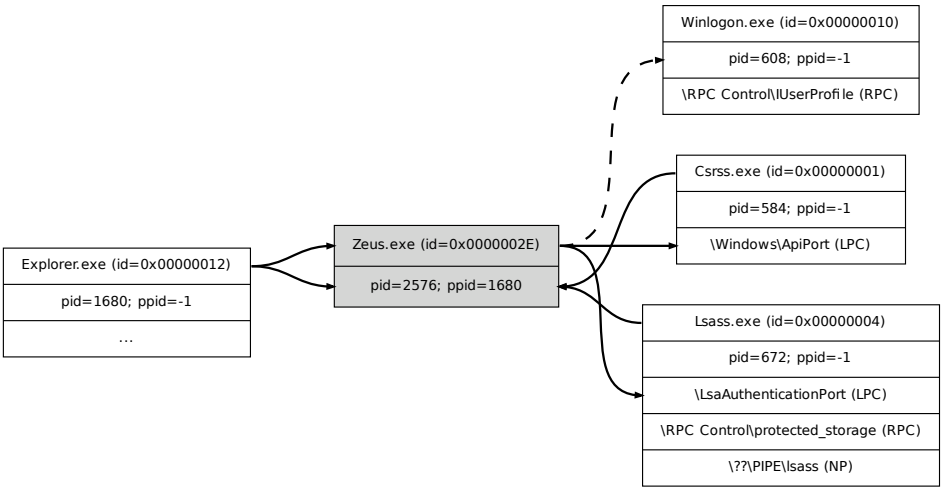
Malware often interacts with other processes when exploiting the host that it infiltrates. Frequently, malicious software injects code into trusted processes so that its illicit operations appear to come from the latter. In this way, the malware has a greater chance of going unnoticed by the victim or antivirus software that may be running on the user's behalf. We deployed PRISON in a virtual environment and executed a number of malware instances to determine how well our system can defend against malicious process interactions. We ran each malware sample two times: first, with PRISON configured in logging mode and second, with the system in filtering mode. This way, we could use PRISON to learn more about the samples' operations and then demonstrate the system's ability to contain the malware.

We executed three malware samples that perform injection of code into the address space of other running processes. The first malware instance was a Zeus bot that injects code into the Winlogon process as part of its installation. The second sample was a Korgo bot that injects a thread into the Windows shell, `Explorer.exe`. The third sample was a suspicious application called YGB Hack Time that injects a DLL called `Itrack.dll` into most running processes.

PRISON successfully blocked all of the malware samples from interacting with other processes. Zeus was contained because it attempted to inject code into `Winlogon.exe`. Figure 5.2 shows the interaction graphs that PRISON automatically generated from the two executions of the Zeus bot; one with PRISON in logging mode, and the other with our system in filtering mode. The logging mode graph (Figure 5.2a)



(a) logging mode



(b) filtering mode

Figure 5.2: Analysis of Zeus bot in logging mode and filtering mode.

shows that Zeus successfully communicates with the Winlogon process and induces it to open a named pipe (called `_AVIRA_2109`). By contrast, with filtering mode enabled, PRISON blocks this interaction (depicted by the dashed arrow in Figure 5.2b), thereby preventing Zeus from completing its installation. Notice that in filtering mode, `Explorer.exe`, `Csrss.exe`, and `Lsass.exe` were still able to communicate with Zeus as they normally do. These three executables are system processes, and these interactions do not violate the filtering policy. Korgo was blocked because a child process that it creates attempted to inject code into `Explorer.exe`. YGB was mitigated because it attempted to set a Windows hook into all running processes, and PRISON filters global Windows hook requests. These results show that malware uses a variety of interaction mechanisms, which underscores the importance of our broad coverage. Furthermore, the results demonstrate that PRISON is an effective tool for both dynamic malware analysis and malware containment.

5.4.2 Evaluating Commercial Security Products

The previous experiment demonstrates that PRISON is able to effectively mitigate the threat of malware that attempts to interact with other processes. A number of commercial host-based security products claim to feature techniques that also accomplish such blocking. We wanted to evaluate these tools to determine the degree to which they are able to filter malicious process interactions, particularly in comparison to PRISON.

Attack Tool Suite. In order to test the commercial security products, we developed a suite of attack tools that exercise a number of the process interaction techniques that we described in Section 5.2. The tools are divided into two classes: inter-process communication tools and injection tools. In particular, the tools that perform inter-process communication leverage the following attacks: One tool sends `WM_SETTEXT` Windows messages to an instance of Notepad to display arbitrary text in its window. Another tool acts as a keylogger that uses the `AttachThreadInput` API to capture keystrokes. A third attack tool uses the `WWW_OpenURL` DDE command to force Internet Explorer (IE) to download a binary on behalf of the attacker from a specified website. This tool also utilizes COM to drive IE (using the `Navigate` method of the `IWebBrowser2` interface) or the Background Intelligent Transfer Service (using `IBackgroundCopyManager` interface methods) to accomplish the binary download. Finally, a set of tools create attack scenarios to exercise communication over shared memory, named pipes, mailslots, and RPCs, respectively. In each case, the tools launch a fictitious (trusted) server that consumes messages that are delivered by an attacker that uses the corresponding communication mechanism.

The goal of the injection tool is to inject an attacker's code into a target process and then execute that code. The tool attempts the actual injection in one of three ways: First, the tool can write code into the target by using the `WriteProcessMemory` API. Second, it can map a file that contains the code into the target's address space (using `MapViewOfFile`). Finally, the tool can use the `SetWindowsHookEx` API to inject a DLL into the target. The next step is to force the target to execute the code. The tool uses three mechanisms for this: `CreateRemoteThread`, the debugging

API, or an asynchronous procedure call (APC). Note that in the case of injection via `SetWindowsHookEx`, the execution occurs automatically, since the entry point of the attacker's DLL will be called when the module is loaded. Otherwise, in order for the attack to be successful, the tool must leverage at least one technique from both stages (i.e., injection and forced execution).

Commercial Security Products. We evaluated the most recent versions of five different commercial security products against our attack tool suite. The products include BitDefender Total Security 2012, Kaspersky Internet Security 2011, McAfee Total Protection 2011, Outpost Internet Security Suite v7.5, and ZoneAlarm Extreme Security 2012. We tested each product in two scenarios. First, we installed the free evaluation version of the products and employed their out-of-the-box configuration. Second, we used the products' configuration options to manually harden them against attacks. While this improved the detection of malicious interactions in some cases, it also frequently led to many unrelated false alarms. Using our attack tool suite, we evaluated both scenarios for each product as well as PRISON.

Summary of Results. Table 5.1 shows the detailed results of the tests, indicating whether an attack was detected (✓) or not (✗). To summarize, we found that BitDefender, Kaspersky, and McAfee offer no process interaction protection in their default configuration (shown in the Std column). After hardening BitDefender and Kaspersky, the detection improved somewhat (the Sec column), but the products still only detected 53% and 41% of the attacks, respectively. McAfee does not provide an option to improve its security posture. Outpost and ZoneAlarm do have security configuration

options, but the settings had no effect on our attacks. In particular, the hardened configuration for these products each matched their out-of-the-box detection rates of 71% and 41%, respectively. PRISON was able to detect and block all of the attacks except for RPC communication over the TCP/IP transport (a detection rate of 94%). However, as we discussed in Section 5.3.2, PRISON ignores communication attempts to endpoints over the network, so we do not consider this attack to be a problem. By contrast, each of the aforementioned commercial tools fails to prevent against infections that abuse interactions with trusted processes. That is, these products are designed to prevent malware from leveraging a "download and execute" style of attack, but they fail to do so when such an attack is laundered through a trusted process.

Table 5.1: Results of interaction tests against security products.

Interaction Type	BitDefender		Kaspersky		McAfee		Outpost		ZoneAlarm		PRISON
	Std	Sec	Std	Sec	Std	Sec	Std	Sec	Std	Sec	
Windows message – Notepad	✗	✗	✗	✗	✗	–	✓	✓	✗	✗	✓
Keylogger	✗	✓	✗	✗	✗	–	✓	✓	✗	✗	✓
DDE – IE	✗	✓	✗	✓	✗	–	✓	✓	✓	✓	✓
COM – IE	✗	✓	✗	✓	✗	–	✓	✓	✓	✓	✓
COM – BITS	✗	✓	✗	✓	✗	–	✓	✓	✗	✗	✓
Shared memory	✗	✗	✗	✗	✗	–	✗	✗	✗	✗	✓
Named pipe	✗	✗	✗	✗	✗	–	✗	✗	✗	✗	✓
Mailslot	✗	✗	✗	✗	✗	–	✗	✗	✗	✗	✓
RPC over named pipe	✗	✗	✗	✗	✗	–	✗	✗	✗	✗	✓
RPC over LPC	✗	✗	✗	✗	✗	–	✗	✗	✗	✗	✓
RPC over TCP/IP	✗	✗	✗	✗	✗	–	✓	✓	✗	✗	✗
Injection – `WriteProcessMemory`	✗	✓	✗	✓	✗	–	✓	✓	✓	✓	✓
Injection – `MapViewOfFile`	✗	✗	✗	✗	✗	–	✓	✓	✗	✗	✓
Injection – `SetWindowsHookEx`	✗	✓	✗	✗	✗	–	✓	✓	✓	✓	✓
Code exec. – `CreateRemoteThread`	✗	✓	✗	✓	✗	–	✓	✓	✓	✓	✓
Code exec. – Debug API	✗	✓	✗	✓	✗	–	✓	✓	✓	✓	✓
Code exec. – APC	✗	✓	✗	✓	✗	–	✓	✓	✓	✓	✓

5.4.3 Completeness

A system that claims to monitor all process interactions must ensure that a process is not able to bypass the analysis by making use of an unknown interface that the system did not anticipate (i.e., the system should be complete). Furthermore, such a system must not introduce false positives by claiming that interactions have occurred that, in fact, did not (i.e., the system should be accurate). In order to test these properties with respect to PRISON, we implemented an *interaction oracle* that does whole-system dynamic taint tracking on the Windows XP platform.

Our oracle tracks all data flow through the operating system. It accomplishes this by tagging all data with a taint label that corresponds to the process to which the data belongs. In this way, when one process interacts with another such that the source passes data to the target, the latter will acquire taint labels that are associated with the former. Therefore, the propagation of taint labels through this system demonstrates all instances in which process interactions occur.

By running PRISON on a host that is also analyzed by the interaction oracle, we can compare the results of the two perspectives. Specifically, we can determine if the taint labels that are propagated by the oracle correspond to interactions that PRISON identifies. If so, this gives us assurance that PRISON is an effective tool for monitoring all of the interactions that occur on the host (given the test cases).

Oracle Implementation

We implemented our taint tracking system as an extension to the Anubis malware analysis sandbox [75]. Anubis itself is built upon QEMU, a powerful virtualization environment that performs processor emulation through dynamic binary translation [12]. This gives our oracle complete control over the guest operating system on a per-instruction basis. To perform data tracking, the oracle leverages the taint propagation facilities that Anubis provides [8]. Thus, our focus was on implementing a taint introduction policy that suits our needs.

The interaction oracle introduces taint into the system at a byte-level granularity. The oracle analyzes each write instruction. When a source operand is tainted, the destination operand is tainted with the same label. If not, the oracle determines if the address of the destination operand is within a specially tracked memory region – namely, a stack, a heap, or a data segment. If this is the case, the oracle taints the destination address with the label that corresponds to the current process. The intuition behind this taint introduction policy is that a process will write its data into the stack, heap, and data segment as part of its normal operation (e.g., when initializing local variables or writing to a dynamically allocated buffer). Thus, the oracle taints all values that a process writes (and that do not derive from already tainted data) with the label that corresponds to this process. Furthermore, taint is introduced when certain events occur. For example, when a process initializes its read-only data segment (`.rdata`) or maps a file into its address space, the oracle taints these memory regions appropriately. This taint policy allows the oracle to accurately track all instances in which data flows from one process to another.

Evaluating PRISON Against the Oracle

We deployed PRISON in a virtualized environment with the interaction oracle running. We again performed the experiments with the attack tool suite as we previously described in Section 5.4.2. We manually compared the perspectives of the two systems in order to evaluate the completeness of our approach.

The oracle's log contains, on a per-process basis, a list of virtual address ranges and the process taint labels that are associated with each region. For each process in the log, we extracted a list of unique taint labels that corresponded to other processes. This list represents the oracle's summary of the processes that interacted with the given process.

We compared the oracle's process interaction list to the output from PRISON's interaction log and attempted to find corresponding entries that explain the interaction. In all cases, we found that the process interaction was accounted for. A few spurious taints were introduced into the analysis, but we manually investigated each of these instances and discovered that they were artifacts of residual taint propagation and should be discounted. For example, in some cases, the oracle reported a few taint labels in a process' address space that corresponded to another process that terminated before the given process began execution. These instances can be safely ignored. Thus, we are confident that PRISON monitors all known process interaction mechanisms.

We also investigated all of the process interactions that were reported in PRISON's interaction log to determine if a corresponding set of taint labels existed in the oracle's log. In all cases, each interaction that was reported by PRISON was also captured by the oracle. This gives us confidence that PRISON does not introduce nonexistent

interactions.

5.4.4 Performance Impact

PRISON is implemented as an extension to the Windows XP kernel, and it must monitor a number of system services that support process interactions. These kernel functions are executed frequently, so it is important to measure the overhead that our system introduces with respect to these invocations.

To this end, we devised three microbenchmarks to evaluate PRISON's impact on process interaction performance. The first two experiments were designed to simulate common inter-process communication scenarios. In particular, we launched an RPC server that listened on an LPC port and named pipe, respectively. Then, a client connected to each server and invoked 100,000 RPC calls. Recall that COM operates over RPC, so these tests encompass a broad range of inter-process communication mechanisms. In the third experiment, we emulated a direct data injection attack. Specifically, a source process allocated a region of memory in the address space of a target process and then executed 1 million calls to `WriteProcessMemory`. Table 5.2 shows the results. The overhead can be attributed to the additional locking and accounting that PRISON performs. This particularly impacts LPC tracking, due to the additional analysis required to identify potential COM endpoints. While the overhead measurements may seem high, we note that these microbenchmarks represent specific interaction operations that comprise a small portion of the host's overall activity; the measurements do not constitute a degradation of the entire system. As such, we believe that the over-

head is acceptable.

Table 5.2: System service overhead (in milliseconds) with PRISON.

Interaction mechanism	Without PRISON	With PRISON	Overhead
RPC over LPC	1834	4194	129%
RPC over named pipe	3098	4209	36%
`WriteProcessMemory`	6421	8748	36%

5.5 Security Analysis

In this section, we analyze the security of our proposed approach to monitoring malicious process interactions. In our threat model, we assume that a malware instance executes as an unprivileged, user-mode process, and it wishes to use an available mechanism to interact with a trusted process in order to carry out malicious actions on its behalf. We acknowledge that our threat model is somewhat weak, because we assume that malicious processes run under a non-administrator user account. However, as we discussed in Section 4.1.2, we believe that this assumption is not unreasonable in the modern computing environment.

The malicious process could attempt to evade the whitelist filtering policy by setting its path and name to match that of a trusted program that is on the whitelist. For this attack to succeed, the malware would need to overwrite a trusted application. Clearly, this is not viable with respect to system processes, since they reside under the privileged `System32` directory. Nonetheless, it may be possible to subvert the policy if the system administrator adds applications in unprivileged paths to the whitelist. In this

case, we could use a stronger mechanism to establish process identity, such as a code identity primitive [166, 58].

It may be argued that whitelisting policies in general are difficult to maintain due to an ever changing software ecosystem. However, we learned from our experiments that although there are many interactions within the operating system, the interactions are primarily limited to standard, trusted system processes (such as `Csrss.exe` and `Explorer.exe`) that are well-known and long-lived. Furthermore, we note that global distribution mechanisms already exist (such as Microsoft Update), which our system could leverage to perform periodic policy maintenance.

Malware could also interact with a target process indirectly. For example, a malicious process may place a DLL on the file system and leverage the `AppInit_DLLs` legacy feature to execute it. We solve this problem by only allowing trusted processes to write to the `AppInit_DLLs` registry value. Additionally, the malware could edit files on disk that trusted processes may read for configuration. We do not currently address this attack, but we are considering ways to augment our system with a file system interaction tainting approach.

5.6 Summary

We presented PRISON, a system that monitors and blocks all malicious process interactions on the Windows XP platform. Our system is implemented as a host-based kernel extension that interposes on inter-process communication among processes and

injection attempts by which one process loads code or data into the address space of another. By dynamically monitoring the interaction behavior among processes on the host, PRISON can protect a user from malware that attempts to leverage a trusted process to carry out its malicious actions. Furthermore, PRISON can output interaction logs and interaction graphs that provide a malware analyst with a concise summary of the interactions that pertain to processes of interest. We have evaluated our approach along a number of axes. The results demonstrate that PRISON is deployable, useful, and efficient.

Chapter 6

Conclusions

Over the last 30 years, malicious software has evolved considerably, becoming ever more sophisticated and destructive. Simple programs that were designed to amuse or annoy their targets have given way to large-scale botnets that steal personal and financial information from their victims. A considerable security community has formed to mitigate the threat of increasingly insidious malware technologies. This has encouraged an arms race between those who hope to protect the users of the Internet and those who seek to profit by exploiting vulnerabilities in this ecosystem.

In this dissertation, we codified the strategies and techniques that attackers and defenders each employ to reach their respective goals. In particular, we presented a classification of the various malware technologies that attackers leverage and an organization of the stages of malware defense. This systematic approach can help the security community to better understand the threat of malware and to focus their defense efforts most

effectively.

A holistic approach to malware defense includes elements from all three of the stages that we outlined, namely, analysis, detection, and response. To this end, in this dissertation, we presented three contributions to malware defense that collectively address these three stages. In particular, we discussed our analysis of the Torpig botnet, and we described two host-based systems, DYMO and PRISON, that address open problems concerning malware detection and malware containment, respectively.

By taking over the Torpig botnet (for ten days), we were able to glean unique insights into the inner-workings of one of the most sophisticated malicious enterprises to date. This analysis motivated our detection efforts, which led us to develop a dynamic code identity primitive. DYMO implements this primitive to provide fine-grained application-based access controls and to furnish provenance information for network connections. Furthermore, our experience with Torpig convinced us that malware containment is an important complement to detection, particularly when attackers attempt to subvert detection by laundering their attacks through trusted processes. This inspired our approach to analyzing and blocking malicious process interactions, as implemented by PRISON.

These contributions advance the state of the art of malware defense. Certainly, the arms race will continue to escalate, and defense techniques will need to be reevaluated and improved in the future. Our work can form the basis of such future efforts, and it can be used in concert with complementary approaches to protecting users from the threat of advanced malware.

Bibliography

[1] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *USENIX Security Symposium*, Aug. 2000.

[2] M. Aiken, M. Fahndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing Process Isolation. In *ACM Memory Systems Performance and Correctness Workshop (MSPC)*, Oct. 2006.

[3] R. Anderson. Cryptography and Competition Policy - Issues with "Trusted Computing". In *Workshop on Economics and Information Security (WEIS)*, May 2003.

[4] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Symposium on Security and Privacy*, May 1997.

[5] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2010.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Wareld. Xen and the Art of Virtualization. In *ACM Symposium on Operating System Principles (SOSP)*, Oct. 2003.

[7] A. Barth, C. Jackson, and C. Reis. Security Architecture of the Chromium Browser. Technical report, Stanford University, Sept. 2008.

[8] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2009.

[9] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. Insights Into Current Malware Behavior. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2009.

[10] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, Apr. 2006.

[11] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, The MITRE Corporation, Mar. 1973.

[12] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference (ATC)*, Apr. 2005.

[13] T. Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. Harper, Nov. 2000.

[14] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. Aug. 2003.

[15] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, The MITRE Corporation, Apr. 1977.

[16] D. Blazakis. Interpreter Exploitation. In *USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2010.

[17] R. Boscovich. Taking Down Botnets: Microsoft and the Rustock Botnet. `http://blogs.technet.com/b/microsoft_on_the_issues/archive/2011/03/18/taking-down-botnets-microsoft-and-the-rustock-botnet.aspx`, Mar. 2011.

[18] S. Burnette. Notice of Termination of ICANN Registrar Accreditation Agreement. `http://www.icann.org/correspondence/burnette-to-tsastsin-28oct08-en.pdf`, Oct. 2008.

[19] A. J. Burstein. Conducting Cybersecurity Research Legally and Ethically. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2008.

[20] M. Castro, M. Costa, J. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-granularity Software Fault Isolation. In *ACM Symposium on Operating System Principles (SOSP)*, Oct. 2009.

[21] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, Aug. 2005.

[22] C. Y. Cho, D. Babic, E. Chul, R. Shin, and D. Song. Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2010.

[23] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, May 2005.

[24] B. Cogswell and M. Russinovich. RootkitRevealer. `http://technet.microsoft.com/sysinternals/`.

[25] F. Cohen. Computer Viruses: Theory and Experiments. In *DOD/NBS Conference on Computer Security*, Sept. 1984.

[26] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, July 2005.

[27] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *International World Wide Web Conference (WWW)*, Apr. 2010.

[28] M. Cova, C. Leita, O. Thonnard, A. Keromytis, and M. Dacier. An Analysis of Rogue AV Campaigns. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2010.

[29] T. Cranton. Cracking Down on Botnets. `http://blogs.technet.com/b/microsoft_on_the_issues/archive/2010/02/24/cracking-down-on-botnets.aspx`, Feb. 2010.

[30] D. Dagon, C. Zou, and W. Lee. Modeling Botnet Propagation Using Time Zones. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2006.

[31] D. Danchev. AVG and Rising signatures update detects Windows files as malware. `http://www.zdnet.com/blog/security/avg-and-rising-signatures-update-detects-windows-files-as-malware/2158`, Nov. 2008.

[32] H. Dang. The Origins of Social Engineering. *McAfee Security Journal*, 2008.

[33] N. Daswani and M. Stoppelman. The Anatomy of Clickbot.A. In *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*, Apr. 2007.

[34] C. R. Davis, J. M. Fernandez, S. Neville, and J. McHugh. Sybil attacks as a mitigation strategy against the Storm botnet. In *International Conference on Malicious and Unwanted Software (MALWARE)*, Oct. 2008.

[35] K. Dunham. Analysis of Sinowal Malicious Code. White paper, VeriSign, Inc., July 2007.

[36] P. Efstathopoulos, M. N. Krohn, S. Vandebogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, M. F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *ACM Symposium on Operating System Principles (SOSP)*, Oct. 2005.

[37] M. Egele, C. Kruegel, and E. Kirda. Dynamic Spyware Analysis. In *USENIX Annual Technical Conference (ATC)*, June 2007.

[38] Eggheads. Eggdrop IRC Bot. `http://www.eggheads.org/`.

[39] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *Computer*, July 2003.

[40] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *European Conference on Computer Systems (EuroSys)*, Apr. 2006.

[41] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. White paper, Symantec Corporation, Nov. 2010.

[42] P. Ferrie and P. Szor. Zmist Opportunities. *Virus Bulletin*, Mar. 2001.

[43] S. Fewer. Reflective DLL Injection. Technical report, Harmony Security, Oct. 2008.

[44] Finjan. How a Cybergang Operates a Network of 1.9 Million Infected Computers. `http://www.finjan.com/SecureTweets/Blog/post/How-a-cybergang-operates-a-network-of-19-million-infected-computers.aspx`, Apr. 2009.

[45] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic Blending Attacks. In *USENIX Security Symposium*, July 2006.

[46] B. Ford and R. Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX Annual Technical Conference (ATC)*, June 2008.

[47] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy*, May 1996.

[48] M. Fossi and E. Johnson. Symantec Report on the Underground Economy. White paper, Symantec Corporation, Nov. 2008.

[49] M. Fossi and E. Johnson. Symantec Global Internet Security Threat Report, Volume XV. White paper, Symantec Corporation, Apr. 2010.

[50] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007.

[51] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *IEEE Symposium on Security and Privacy*, May 1999.

[52] F. C. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *European Symposium on Research in Computer Security (ESORICS)*, Sept. 2005.

[53] L. Garber. Melissa Virus Creates a New Type of Threat. *Computer*, June 1999.

[54] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, May 2007.

[55] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *ACM Symposium on Operating System Principles (SOSP)*, Oct. 2003.

[56] T. Garnkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2003.

[57] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital Distributed System Security Architecture. In *National Computer Security Conference*, Jan. 1989.

[58] B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna. Dymo: Tracking Dynamic Code Identity. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2011.

[59] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *USENIX Security Symposium*, July 1996.

[60] R. P. Goldberg. Survey of Virtual Machine Research. *Computer*, June 1974.

[61] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, May 2008.

[62] T. C. Group. TPM Main Specification version 1.2, July 2007.

[63] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *USENIX Security Symposium*, Aug. 2008.

[64] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *USENIX Security Symposium*, Aug. 2007.

[65] P. Guehring. Concepts against Man-in-the-Browser Attacks. `http://www2.futureware.at/svn/sourcerer/CAcert/SecureClient.pdf`, Sept. 2006.

[66] F. Guo, P. Ferrie, and T. Chiueh. A Study of the Packer Problem and Its Solutions. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, Apr. 2008.

[67] N. Hardy. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, Oct. 1988.

[68] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison Wesley Professional, 2005.

[69] T. Holz, M. Engelberth, and F. Freiling. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. In *European Symposium on Research in Computer Security (ESORICS)*, Sept. 2009.

[70] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling. Measuring and Detecting Fast-Flux Service Networks. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2008.

[71] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2008.

[72] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *USENIX Windows NT Symposium*, July 1999.

[73] G. Hunt, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, and T. Wobber. Sealing OS Processes to Improve Dependability and Safety. In *European Conference on Computer Systems (EuroSys)*, Mar. 2007.

[74] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, Apr. 2007.

[75] International Secure Systems Lab. Anubis: Analyzing Unknown Binaries. `http://anubis.iseclab.org/`.

[76] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-Operating Systems: A New Approach to Application Security. In *ACM SIGOPS European Workshop*, July 2002.

[77] D. Jackson. Untorpig. `http://www.secureworks.com/research/tools/untorpig/`, Oct. 2008.

[78] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y. Wang, and E. H. Spafford. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. In *International Conference on Distributed Computing Systems (ICDCS)*, July 2006.

[79] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007.

[80] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy*, May 2006.

[81] P. Kamp and R. N. M. Watson. Jails: Conning the omnipotent root. In *System Administration and Networking Conference (SANE)*, May 2000.

[82] B. B. Kang, E. Chan-Tin, C. P. Lee, J. Tyra, H. J. Kang, C. Nunnery, Z. Wadler, G. Sinclair, N. Hopper, D. Dagon, and Y. Kim. Towards Complete Node Enumeration in a Peer-to-Peer Botnet. In *ACM Symposium on Information, Computer and Communication Security (ASIACCS)*, Mar. 2009.

[83] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *ACM Workshop on Rapid Malcode (WORM)*, Nov. 2007.

[84] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.

[85] C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, and S. Savage. The Heisenbot Uncertainty Problem: Challenges in Separating Bots from Chaff. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2008.

[86] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-Scale Botnet Detection and Characterization. In *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*, Apr. 2007.

[87] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *IEEE Symposium on Security and Privacy*, Apr. 1984.

[88] Kaspersky Lab. Kaspersky Lab provides its insights on Stuxnet worm. `http://www.kaspersky.com/news?id=207576183`, Sept. 2010.

[89] K. Kasslin and E. Florio. Your Computer is Now Stoned (...Again!). The Rise of MBR Rootkits. In *Virus Bulletin Conference (VB)*, Oct. 2008.

[90] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 1994.

[91] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, Aug. 2004.

[92] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting Malicious Code by Model Checking. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2005.

[93] S. T. King and P. M. Chen. Backtracking Intrusions. In *ACM Symposium on Operating System Principles (SOSP)*, Oct. 2003.

[94] S. T. King, P. M. Chen, Y. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In *IEEE Symposium on Security and Privacy*, May 2006.

[95] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching Intrusion Alerts through Multi-Host Causality. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2005.

[96] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based Spyware Detection. In *USENIX Security Symposium*, July 2006.

[97] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-side Solution for Mitigating Cross-Site Scripting Attacks. In *ACM Symposium on Applied Computing (SAC)*, Apr. 2006.

[98] I. Kirillov, P. Chase, D. Beck, and R. Martin. Malware Attribute Enumeration and Characterization (MAEC). White paper, The MITRE Corporation, Feb. 2010.

[99] J. Kirk. Dutch Team Up With Armenia for Bredolab Botnet Take Down. `http://www.networkworld.com/news/2010/102610-dutch-team-up-with-armenia.html`, Oct. 2010.

[100] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *USENIX Security Symposium*, Aug. 2009.

[101] B. Krebs. Host of Internet Spam Groups Is Cut Off. `http://www.washingtonpost.com/wp-dyn/content/article/2008/11/12/AR2008111200658.html`, Nov. 2008.

[102] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the Spam Campaign Trail. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2008.

[103] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *European Symposium on Research in Computer Security (ESORICS)*, Oct. 2003.

[104] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Applications Conference (ACSAC)*, Dec. 2004.

[105] S. Kumar and E. H. Spafford. A Generic Virus Scanner in C++. In *Annual Computer Security Applications Conference (ACSAC)*, Dec. 1992.

[106] B. W. Lampson. Protection. In *Princeton Conference on Information Sciences and Systems*, Jan. 1971.

[107] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, Oct. 1973.

[108] W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, Dec. 1992.

[109] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[110] J. Leyden. Conficker botnet growth slows at 10m infections. `http://www.theregister.co.uk/2009/01/26/conficker_botnet/`, Jan. 2009.

[111] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2003.

[112] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *USENIX Security Symposium*, July 2008.

[113] D. Mandelin. An Overview of TraceMonkey. http://hacks.mozilla.org/2009/07/tracemonkey-overview/, July 2009.

[114] J. Markoff. Before the Gunfire, Cyberattacks. http://www.nytimes.com/2008/08/13/technology/13cyber.html, Aug. 2008.

[115] J. Markoff. Cyberattacks Jam Government and Commercial Web Sites in U.S. and South Korea. http://www.nytimes.com/2009/07/10/technology/10cyber.html, July 2009.

[116] J. Markoff. A Silent Attack, but Not a Subtle One. http://www.nytimes.com/2010/09/27/technology/27virus.html, Sept. 2010.

[117] J. Markoff. Evidence Found for Chinese Attack on Google. http://www.nytimes.com/2010/01/20/technology/20cyber.html, Jan. 2010.

[118] J. Markoff and D. E. Sanger. In a Computer Worm, a Possible Biblical Clue. http://www.nytimes.com/2010/09/30/world/middleeast/30worm.html, Sept. 2010.

[119] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Annual Computer Security Applications Conference (ACSAC)*, Dec. 2007.

[120] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security Symposium*, Aug. 2006.

[121] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *European Conference on Computer Systems (EuroSys)*, Apr. 2008.

[122] R. McMillan. Conficker group says worm 4.6 million strong. `http://www.cw.com.hk/content/conficker-group-says-worm-46-million-strong`, Apr. 2009.

[123] J. McMullan and A. Rege. Cyberextortion at Online Gambling Sites: Criminal Organization and Legal Challenges. *Gaming Law Review*, Dec. 2007.

[124] Microsoft Corporation. BitLocker Drive Encryption. `http://windows.microsoft.com/en-US/windows7/products/features/bitlocker`.

[125] Microsoft Corporation. Malicious Software Removal Tool. `http://www.microsoft.com/security/malwareremove/`.

[126] Microsoft Corporation. Mandatory Integrity Control. `http://msdn.microsoft.com/en-us/library/bb648648(VS.85).aspx`.

[127] Microsoft Corporation. New UAC Technologies for Windows Vista. `http://msdn.microsoft.com/en-us/library/bb756960.aspx`.

[128] Microsoft Corporation. A Detailed Description of the Data Execution Prevention (DEP) Feature. http://support.microsoft.com/kb/875352, Sept. 2006.

[129] Microsoft Corporation. Description of the Software Restriction Policies in Windows XP. http://support.microsoft.com/kb/310791, Sept. 2009.

[130] M. Miller, K. Yee, and J. S. Shapir. Capability Myths Demolished. Technical Report SRL2003-02, Systems Research Laboratory, Johns Hopkins University, 2003.

[131] Miniwatts Marketing Group. World Internet Usage Statistics. http://www.internetworldstats.com/stats.htm, Mar. 2011.

[132] K. D. Mitnick and W. L. Simon. *The Art of Deception: Controlling the Human Element of Security*. Wiley, Oct. 2002.

[133] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy Magazine*, Jan. 2003.

[134] D. Moore, C. Shannon, and J. Brown. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *ACM SIGCOMM Internet Measurement Workshop (IMW)*, Nov. 2002.

[135] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, May 2007.

[136] A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In *Annual Computer Security Applications Conference (ACSAC)*, Dec. 2007.

[137] N. Murilo and K. Steding-Jessen. chkrootkit. `http://www.chkrootkit.org/`.

[138] K. Natvig. Sandbox Technology Inside AV Scanners. In *Virus Bulletin Conference (VB)*, Sept. 2001.

[139] Net Applications.com. Global Market Share Statistics. `http://marketshare.hitslink.com/`, Sept. 2011.

[140] P. G. Neumann and R. J. Feiertag. PSOS Revisited. In *Annual Computer Security Applications Conference (ACSAC)*, Dec. 2003.

[141] J. Nick L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*, Aug. 2004.

[142] G. Ollmann. Caution Over Counting Numbers in C&C Portals. `http://blog.damballa.com/?p=157`, Apr. 2009.

[143] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *IEEE Symposium on Security and Privacy*, May 2010.

[144] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Journal of Computer Networks*, Dec. 1999.

[145] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. M. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet Judo: Fighting Spam with Itself. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2010.

[146] M. Polychronakis, P. Mavrommatis, and N. Provos. Ghost turns Zombie: Exploring the Life Cycle of Web-based Malware. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2008.

[147] P. Porras, H. Saidi, and V. Yegneswaran. A Foray into Conficker's Logic and Rendezvous Points. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2009.

[148] D. Price and A. Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *USENIX Large Installation Systems Administration Conference (LISA)*, Nov. 2004.

[149] N. Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, Aug. 2003.

[150] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symposium*, July 2008.

[151] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*, Apr. 2007.

[152] M. A. Rajab, F. Monrose, A. Terzis, and N. Provos. Peeking through the Cloud: DNS-Based Estimation and its Applications. In *International Conference on Applied Cryptography and Network Security (ACNS)*, June 2008.

[153] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *ACM Internet Measurement Conference (IMC)*, Oct. 2006.

[154] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. My Botnet is Bigger Than Yours (Maybe, Better Than Yours): Why Size Estimates Remain Challenging. In *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*, Apr. 2007.

[155] A. Ramachandran, K. Bhandankar, M. B. Tariq, and N. Feamster. Packets with Provenance. Technical Report GT-CS-08-02, Georgia Institute of Technology, 2008.

[156] A. Ramachandran, N. Feamster, and D. Dagon. Revealing Botnet Membership Using DNSBL Counter-Intelligence. In *USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, July 2006.

[157] A. Ramachandran, Y. Mundada, M. B. Tariq, and N. Feamster. Securing Enterprise Networks Using Traffic Tainting (Poster). In *ACM Special Interest Group on Data Communication (SIGCOMM)*, Aug. 2009.

[158] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *European Conference on Computer Systems (EuroSys)*, Apr. 2009.

[159] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2008.

[160] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX Large Installation Systems Administration Conference (LISA)*, Nov. 1999.

[161] P. Royal. On the Kraken and Bobax Botnets. White paper, Damballa, Apr. 2008.

[162] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Annual Computer Security Applications Conference (ACSAC)*, Dec. 2006.

[163] RSA FraudAction Lab. One Sinowal Trojan + One Gang = Hundreds of Thousands of Compromised Accounts. `http://www.rsa.com/blog/blog_entry.aspx?id=1378`, Oct. 2008.

[164] J. Rutkowska. System Virginity Verifier: Defining the Roadmap for Malware Detection on Windows System. In *Hack In The Box Security Conference*, Sept. 2005.

[165] J. Rutkowska. Subverting Vista Kernel for Fun and Profit. In *Black Hat Briefings*, Aug. 2006.

[166] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security Symposium*, Aug. 2004.

[167] J. H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, July 1974.

[168] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, Sept. 1975.

[169] K. Scarfone and P. Mell. Guide to Intrusion Detection and Prevention Systems (IDPS). Special Publication 800-94, National Institute of Standards and Technology, Feb. 2007.

[170] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. In *ACM Symposium on Operating System Principles (SOSP)*, Oct. 2005.

[171] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2007.

[172] C. Shannon and D. Moore. The Spread of the Witty Worm. *IEEE Security and Privacy Magazine*, July 2004.

[173] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *ACM Symposium on Operating System Principles (SOSP)*, Dec. 1999.

[174] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy*, May 2009.

[175] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee. Eureka: A Framework for Enabling Static Malware Analysis. In *European Symposium on Research in Computer Security (ESORICS)*, Oct. 2008.

[176] M. Shields. Trojan virus steals banking info. `http://news.bbc.co.uk/2/hi/technology/7701227.stm`, Oct. 2008.

[177] M. Silbey and P. Brundrett. Understanding and Working in Protected Mode Internet Explorer. White paper, Microsoft Corporation, Jan. 2006.

[178] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[179] R. Skrenta. The joy of the hack. `http://www.skrenta.com/2007/01/the_joy_of_the_hack.html`, June 2007.

[180] Sophos. Security at risk as one third of surfers admit they use the same password for all websites, Sophos reports. `http://www.sophos.com/pressoffice/news/articles/2009/03/password-security.html`, Mar. 2009.

[181] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *USENIX Security Symposium*, Aug. 2002.

[182] A. E. Stepan. Defeating Polymorphism: Beyond Emulation. In *Virus Bulletin Conference (VB)*, Oct. 2005.

[183] J. Stewart. Storm Worm DDoS Attack. `http://www.secureworks.com/research/threats/storm-worm/`, Feb. 2007.

[184] B. Stone-Gross, R. Abman, R. Kemmerer, C. Kruegel, D. Steigerwald, and G. Vigna. The Underground Economy of Fake Antivirus Software. In *Workshop on Economics and Information Security (WEIS)*, June 2011.

[185] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2009.

[186] B. Stone-Gross, M. Cova, B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna. Analysis of a Botnet Takeover. *IEEE Security and Privacy Magazine*, Jan. 2011.

[187] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX Annual Technical Conference (ATC)*, June 2001.

[188] P. Szor. Olivia. *Virus Bulletin*, June 1997.

[189] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, Feb. 2005.

[190] A. Tan. Flawed Symantec update cripples Chinese PCs. `http://news.cnet.com/Flawed-Symantec-update-cripples-Chinese-PCs/2100-1002_3-6186271.html`, May 2007.

[191] The MITRE Corporation. Common Vulnerabilities and Exposures (CVE). `http://cve.mitre.org/`.

[192] V. Thomas and N. Jyoti. Defeating IRC Bots on the Internal Network. *Virus Bulletin*, Feb. 2007.

[193] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, Aug. 1984.

[194] N. Thornburg. The Invasion of the Chinese Cyberspies. http://www.time.com/time/magazine/article/0,9171,1098961-1,00.html, Aug. 1995.

[195] United States Department of Justice. Department of Justice Takes Action to Disable International Botnet. http://www.justice.gov/opa/pr/2011/April/11-crm-466.html, Apr. 2011.

[196] F. Valeur, G. Vigna, C. Kruegel, and R. A. Kemmerer. A Comprehensive Approach to Intrusion Detection Alert Correlation. *IEEE Transactions on Dependable and Secure Computing*, July 2004.

[197] W. Venema. Isolation Mechanisms for Commodity Applications and Platforms. Technical Report RC24725, IBM Research, Jan. 2009.

[198] VeriSign iDefense Intelligence Operations Team. The Russian Business Network: Rise and Fall of a Criminal ISP. White paper, VeriSign, Inc., Mar. 2008.

[199] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, May 2001.

[200] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *ACM Symposium on Operating System Principles (SOSP)*, Dec. 1993.

[201] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security Symposium*, Aug. 2009.

[202] K. Wang and S. J. Stolfo. Anomalous Payload-Based Network Intrusion Detection. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2004.

[203] Y. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. In *International Conference on Dependable Systems and Networks (DSN)*, June 2005.

[204] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2009.

[205] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: practical capabilities for UNIX. In *USENIX Security Symposium*, Aug. 2010.

[206] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A Taxonomy of Computer Worms. In *ACM Workshop on Rapid Malcode (WORM)*, Oct. 2003.

[207] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *USENIX Security Symposium*, Aug. 2004.

[208] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy Magazine*, Mar. 2007.

[209] T. Wobber, A. R. Yumerefendi, M. Abadi, A. Birrell, and D. R. Simon. Authorizing Applications in Singularity. In *European Conference on Computer Systems (EuroSys)*, Mar. 2007.

[210] J. Wolf. Technical Details of Srizbi's Domain Generation Algorithm. http://blog.fireeye.com/research/2008/11/technical-details-of-srizbis-domain-generation-algorithm.html, Nov. 2008.

[211] P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda. Automatically Generating Models for Botnet Detection. In *European Symposium on Research in Computer Security (ESORICS)*, Sept. 2009.

[212] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*, May 2009.

[213] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007.

[214] A. Young and M. Yung. Cryptovirology: Extortion-Based Security Threats and Countermeasures. In *IEEE Symposium on Security and Privacy*, May 1996.

[215] A. Zeigler. IE8 and Loosely-Coupled IE (LCIE). http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx.

[216] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazires. Making Information Flow Explicit in HiStar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[217] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing Distributed Systems with Information Flow Control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2008.

[218] K. Zetter. Google Hack Attack Was Ultra Sophisticated, New Details Show. `http://www.wired.com/threatlevel/2010/01/operation-aurora/`, Jan. 2010.

[219] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System Support for Derived Data Management. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Mar. 2010.

[220] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, I. Osipkov, G. Hulten, and J. D. Tygar. Characterizing Botnets from Email Spam Records. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2008.