PROTECTING CONFIDENTIAL INFORMATION FROM MALICIOUS SOFTWARE

by

Kevin R. Borders

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Doctoral Committee:

Professor Atul Prakash, Chair
Professor Peter M. Chen
Professor Myron P. Gutmann
Assistant Professor Morley Mao
Associate Professor Patrick D. McDaniel, Penn State University

المنارة للاستشارات

www.manaraa.com

© Kevin R. Borders 2009
All Rights Reserved

www.manaraa.com

ACKNOWLEDGEMENTS

I am indebted to everyone who has taken part in my education starting from an early age. In chronological order, I would like to thank my parents, Gwen and John, for providing an intellectually stimulating environment at home and making sure I was always challenged in school.

There are several teachers who have had an impact on me throughout grade school and high school. I would like to thank Jon Sevald for teaching me to think creatively and offering nothing but words of encouragement when I said that I wanted to get a Ph. D. at such a young age. I would also like to thank Mary Kay Glazek and my other English teachers at Roeper for helping me become a good writer. I am thankful for Chris Bradley, who taught me everything I know about math. Pat Vance and Dennis King were the best science instructors one could have asked for. They taught me the fundamentals of problem solving and experimentation that are invaluable in any field of science.

Before working at the Red Team, I knew nothing about computer security. I would like to thank everyone there, especially Justin, Diana, Mike, Paul, and Shawn, for exposing me to such an exciting area of computer science.

I am grateful to have such a loving and supportive fiancée, Jennifer Cendrowski, who has always valued education and encouraged me to pursue my Ph. D.

Many of my undergraduate courses at the University of Michigan were excellent, but Professor Brian Noble's operating systems class and Professor Mark Brehob's computer architecture class were exceptionally inspiring, and I am thankful to them for making education a high priority.

Throughout graduate school, feedback and collaboration with peers has helped tremendously. I would like to thank co-authors who I have had the pleasure of working with: Xin Zhao, Laura Falk, Patrick Traynor, Kevin Butler, and William Enck.

I greatly appreciate the efforts of my thesis committee members not only in helping with this dissertation, but also throughout school. I would like to thank Professor Peter Chen for attending nearly all of my practice talks and asking tough questions that have influenced the

المنارة للاستشارات

www.manaraa.com

direction of my research. I want to thank Professor Myron Gutmann for providing me input on the Storage Capsule research from the most important perspective: that of the user. I am grateful to Professor Morley Mao for affording me the opportunity to help teach her networking course. Finally, I would like to thank Professor Patrick McDaniel for always giving me great feedback and reviewing several papers on short notice, including the original Web Tap paper.

Last, I want to thank my advisor, Professor Atul Prakash, without whom I may not have pursued a Ph. D. After completing a project for his class, he worked with me day and night to transform the research into a real publication. Following the initial project, he welcomed me into his group and continued to support all of the research ideas – the more farfetched the better, it seems – over the next several years. I am also grateful that he gave me the opportunity to pursue commercialization of the original Web Tap concept, the experience from which has led to a large portion of the results in this thesis. Finally, I would like to thank Professor Prakash for imparting his wisdom and teaching me to be a successful researcher over the past five years, which was time very well spent.

المنارة للاستشارات
www.manaraa.com

TABLE OF CONTENTS

CHAPTER

المنارة للاستشارات

www.manaraa.com

المنارة للاستشارات
www.manaraa.com

LIST OF FIGURES

Figure

المنارة للاستشارات
www.manaraa.com

المنارة للاستشارات

www.manaraa.com

LIST OF TABLES

Table

www.manaraa.com

CHAPTER 1

INTRODUCTION

1.1 Overview

Confidentiality is one of the cornerstones of information security. Ensuring confidentiality has been a major challenge over the past half-century as more information moves from paper and spoken word into digital form. Computers are immensely complex in comparison to their paper counterparts, leading to a plethora of previously unseen security vulnerabilities. Even more problematic is the constant change and increased interaction between computing systems, driven by the need for new functionality. Today, we are far from solving the problem of protecting confidential digital information, as is evidenced by an endless stream of high-profile information leaks [AP05, Katayama03, Reuters06].

One common threat to confidentiality comes from a lack of integrity. When computers load programs and files from low-integrity sources, such as the Internet, they are exposed to hackers, viruses, and other security threats. Traditional methods for maintaining confidentiality involve protecting software that handles sensitive information from low-integrity data, and checking that such software does not leak secret information. Unfortunately, this is in direct opposition to the trend towards increased interaction between computer systems, which is motivated by collaborative applications and the Internet. These opposing objectives force a tradeoff between utility and security when determining the size of the set of systems that must have high integrity, also known as the trusted computing base (TCB). A true high-integrity computing system would be detrimental to productivity, and a completely open system would offer no integrity guarantees.

Some secure systems try to balance the need for integrity and usability. Trusted computing platforms such as Terra [Garfinkel03a] and trusted boot [Sailer04] verify software integrity at startup and when running new applications. This guarantees that the system will start in a clean state and that it will only execute trusted programs. Despite its benefits, however,

المنارة للاستشارات
www.manaraa.com

trusted computing suffers from a number of problems. First, verifying every binary makes installing and managing software much more cumbersome. This is one of the reasons why trusted computing platforms are not widely used. Trusted computing also assumes that new software installed by the user does not contain Trojan horse malware, which is not always the case. While trusted computing platforms do help protect confidentiality, they are still vulnerable to attack.

The security systems presented in this thesis take an alternate approach to preserving confidentiality. Instead of trying to protect computers from malicious inputs, their goal is to make it harder for computers that have been compromised with malicious software to leak sensitive information. This approach is more widely applicable than integrity-preserving solutions, as a huge portion of personal computers today – 100 to 150 million by one estimate [Weber07] – are already infected with malware, and many more are vulnerable. Protecting confidentiality on end hosts is essential, even in the presence of malicious software, as they are the primary portal for creating, viewing, and editing nearly all sensitive digital information.

A number of other solutions exist that help protect confidentiality in low-integrity systems. One example is mandatory access control (MAC), which is used by Security-Enhanced Linux (SELinux) [NSA09]. MAC can control the flow of sensitive data with policies that mark entities that read secret information as secret, and then prevent those entities from communicating over the network. This policy set achieves the goal of preventing leaks, but it also prevents most useful applications from running. Furthermore, correct policy configuration with SELinux can be complicated and difficult even for expert users. A different embodiment of the same principle can be seen in an "air gap" separated network. In an air gap network, a small group of computers is physically separated from the external network, thus completely preventing information flow to that network. While this is also an effective way to protect confidentiality, computers in air gap networks cannot connect to the Internet, which greatly limits their utility. In general, current flow-control mechanisms that block low-integrity systems from leaking data are severely detrimental to usability. This is reinforced by the unpopularity of both mandatory access control with strict information flow policies and air gap networks, which are rarely used outside of protecting classified government information.

Although it is difficult to provide guaranteed confidentiality for low-integrity computers, various network monitoring systems try to combat different threats to confidentiality. Intrusion detection systems (IDSs) aim to detect nefarious network activity [IBM09a, Paxson98, Roesch99]. However, IDSs only provide integrity protection by detecting malicious software. They do not detect leaks themselves and do not help against insider leaks. One form of direct confidentiality protection is a web content filter, which blocks access to unwanted web servers

للاستشارات المنارة
www.manaraa.com

[OpenDNS09, Websense09]. However, content filters are not effective against a smart adversary who posts information to a legitimate website, such as WikiPedia.org, and retrieves it from another location. Data loss prevention (DLP) systems are specifically designed to detect and block sensitive information that is flowing over the network [RSA07, Vontu09]. Unfortunately, they only examine the content of network traffic and are unable to detect encrypted or obfuscated leaks.

The host-level security systems presented in this thesis aim to provide the same level of security as mandatory access control, but are compatible with standard operating systems and applications. The goal of network monitoring systems presented in this thesis is to deliver better protection against a wider array of threats than current solutions. The specific contributions of this work include a host-based system for securely editing sensitive documents on a low-integrity machine, a network monitoring system that identifies traffic from different web applications, and techniques for precisely quantifying network-based information leaks. A combination of host- and network-based security mechanisms provides multi-tiered protection that can take advantage of increased control on the host and enterprise-wide visibility at the network edge. All of the security systems presented here operate under the assumption that end hosts may be compromised with malicious software, and do they not require assistance from secure hardware. The network-based leak measurement techniques further assume that clients may be under the control of inside attackers.

1.2 Host-Level Confidentiality Protection

The first security system that we discuss in Chapter 3 is the Storage Capsule system. The goal of Storage Capsules is to allow users to securely edit confidential documents with a computer that has been compromised by malicious software. Storage Capsules assume that the user, hardware, and some low-level software components are trusted. However, they do not rely on the user's main operating system or applications to maintain any integrity. Storage Capsules protect sensitive information even if the operating system and applications accessing the data are compromised with malicious software.

From the user's perspective, Storage Capsules are similar to other encrypted file containers, such those provided by various compression utilities [Roshal09, Winzip09] and encryption tools [TrueCrypt09]. The primary difference comes when the user opens a Storage Capsule. The system will disable network and persistent disk output while the user has access to the decrypted Storage Capsule contents. After the user has finished editing the Storage Capsule,

المنارة للاستشارات
www.manaraa.com

the all changes that have been made, except those to the Capsule itself, will be erased by reverting the system to its original state. The goal is to prevent malicious software on the computer from leaking information. Traditional encrypted file containers, in comparison, completely expose sensitive plain-text to any malware running on the computer. The user could disable the network adapter while accessing sensitive data, but malware could still record information and send it out at a later time.

In the Storage Capsule architecture, a trusted component sits below the primary operating system, where it can mediate device access. The Capsule system uses keyboard escape sequences, which are intercepted by the trusted component, to transition between normal mode and secure editing mode. The process of editing an encrypted Storage Capsule can be broken down into the following steps:

1. The user opens a Capsule with an application that notifies the trusted component to enter secure editing mode.
2. The application asks the user to press a key escape sequence, which the trusted component will trap. The component will also verify that it has been notified about a transition.
3. The Capsule system saves the primary operating system's state, disables network and device output, and then informs the user that it is safe to decrypt and begin editing the Storage Capsule.
4. The user presses a second key escape sequence. This causes the trusted component to save changes to the Storage Capsule. It then discards all other changes and restores the primary OS back to a snapshot of its original state.

This editing process enables the user to make changes to a Storage Capsule with a compromised computer, save the changes, and then store or transmit the Capsule on an insecure medium. Storage Capsules also allow the user to read encrypted files that come from low-integrity sources, such as e-mail, without having to worry about whether a file will compromise the computer. Even if it does, it will not affect the confidentiality of any files in a Storage Capsule.

Like traditional mandatory access control systems, Storage Capsules are also prone to information leakage via covert channels. If the primary operating system is able to affect the computer's state in any way that is detectable following a snapshot restoration, then it can leak data. The Capsule system makes an effort to minimize the scope and severity of covert channels. It mitigates timing channels by only leaving secure editing mode at the request of the user via a secure keyboard input channel. This prevents a compromised primary OS from controlling the transition time. Storage Capsules also have a fixed size, and are completely re-encrypted every time their cipher-text is accessed by the primary OS outside of secure mode. This prevents

المنارة للاستشارات
www.manaraa.com

malware from manipulating Storage Capsule contents or attributes to leak information. There are also many lower-level covert channels, such as CPU micro-architecture state, that are difficult to block. This thesis makes an effort to enumerate and suggest countermeasures for many of these covert channels.

There are some scenarios where Storage Capsules cannot protect sensitive information. One such case is when the application that handles sensitive data requires network interaction to function properly. For example, Storage Capsules cannot protect account numbers while accessing an online banking service or purchasing an item online with a credit card. Storage Capsules are also incompatible with mobile devices that use light-weight operating systems and cannot support virtual machines. Finally, Storage Capsules do not protect against a compromised virtual machine monitor, compromised hardware, or a malicious user. They operate under the assumption that the user is helping to protect sensitive information and the computer is physically secure. In scenarios where these assumptions do not hold, the network monitoring systems presented in this thesis are a better solution for protecting confidentiality.

1.3 Network-Level Confidentiality Protection

The second part of this thesis focuses on methods for protecting confidentiality that involve monitoring network traffic. Being able to detect breaches of confidentiality, and of integrity, at the network level is important because an attacker may bypass host-based protection mechanisms and subvert individual systems on a network. One way this can happen is if somebody brings a mobile device into an enterprise from an outside environment with more lax security. An even more serious threat is that of an inside attacker who breaks host-based security and sends confidential information out over the network. The network-based protection systems presented in this thesis are designed to handle threats from completely unmanaged clients and clients under the control of malicious insiders.

In Chapter 4, we explore techniques for detecting and differentiating traffic from various web applications. Being able to identify programs by looking at their network traffic allows administrators to quickly respond to confidentiality threats. The most direct threat would be a spyware program whose purpose is stealing sensitive information. However, legitimate programs can also threaten confidentiality. File sharing software may download other programs that come with Trojan horse malware. Instant messaging clients make it easy for users to send files or other sensitive data over the Internet. Detecting the presence of such unwanted programs benefits overall network security.

المنارة للاستشارات
www.manaraa.com

Every application that accesses the Internet has a traffic profile that includes its request formatting, request timing, and the servers with which it communicates. There are particular fields in web requests whose values are specific to the application that generated each request. Grouping requests according to these fields helps classify traffic coming from different applications. Automated processes that generate web traffic also exhibit vastly different timing characteristics than standard human web browsing. Looking at the delay between requests to the same server, their time of day, and the overall request regularity uncovers timer-driven requests and other automated activity. The timing analysis techniques, combined with request format processing, yield low-level alerts detailing web activity from different programs. These alerts are later filtered with a whitelist, which maps them to profiles of specific applications.

This thesis next examines methods for precisely measuring information flow in outbound web traffic in Chapter 5. While one can measure raw web traffic bandwidth, it is usually much larger than the true amount of information conveyed by the traffic. The key insight here is that large portions of web requests are fixed by the protocol or repeated from previous network messages. The measurement algorithms will construct a representation of expected web traffic, and then compute the difference, in terms of information content, between the actual and the expected requests. This process is similar to a compression algorithm that constructs a distribution of expected strings and then distills the actual strings down to their true information content (*entropy*) with respect to the distribution. The resulting byte measurement represents the amount of information needed to reconstruct the request stream, which is a sound upper limit on how much data the client could have leaked, including covert channels. For typical web traffic, this number is two to three orders of magnitude smaller than a naïve raw measurement. The precise bandwidth computations facilitate identification of true network-based leaks by reducing measurements for benign traffic.

The final part of this thesis involves processing low-level information about web application activity and outbound bandwidth utilization. Most request formatting, timing, and bandwidth anomalies are caused by legitimate applications. To filter out alert information from legitimate programs, we employ a whitelist, which contains network activity profiles for known applications. Alerts that remain after whitelist processing indicate suspicious activity. As part of this work, we generated whitelist entries for over 500 legitimate applications during a two-year test deployment in a corporate network. Based on enterprise deployment experience, this thesis outlines a systematic approach for generating new whitelist entries that minimizes the risk of opening a backdoor for attackers or inadvertently trusting malicious software. The process for updating the whitelist is designed to be straightforward enough for security analysts to add new

المنارة للاستشارات
www.manaraa.com

entries without the help of an engineer. Based on the corporate test deployment, we show that the network monitoring systems presented in this thesis effectively detect threats to confidentiality, and have a reasonable overhead for alert analysis and whitelist maintenance.

1.4 Contributions

1.4.1 Storage Capsules

This thesis presents the design, implementation, and evaluation of a system for protecting confidential files with Storage Capsules. Storage Capsules allow safe access to sensitive files from a normal operating system with standard applications. The Capsule system is able to switch modes within one OS rather than requiring separate operating systems or processes for different modes. Storage Capsules provide the same guarantees as traditional mandatory access control, but are compatible with existing software. This thesis also makes contributions in the understanding of covert channels in such a system. In particular, it looks at how virtualization technology can create new covert channels and how previously explored covert channels behave differently when the threat model is a low-security virtual machine running after a high-security virtual machine, on the same computer, rather than two processes running side-by-side.

1.4.2 Detecting Web Applications

Programmatic web requests differ significantly from those driven by human input. This work explores three aspects of web requests that make it possible to identify whether they come from a network application, and, if so, which one. Network application requests often have unique formatting that immediately identifies them. Applications that use generic formatting or emulate that of a web browser can be further differentiated by examining the regularity of their requests, the delay times in between individual requests, and the time of day at which requests occur. These aspects of web traffic allow us to detect the presence of different web applications, including malware and unwanted programs, solely by examining network activity.

1.4.3 Quantifying Information Leaks

A large portion of typical outbound web traffic is repeated or constrained by the protocol. As such, it does not contain any information from the client. Filtering out this constrained data helps to isolate potential information leaks in *unconstrained* traffic. This thesis explores original methods for determining what portion of outbound web request data can be discounted in this manner. The leak measurement techniques involve computing expected headers, links, and form

المنارة للاستشارات
www.manaraa.com

fields by processing prior requests and server responses. In addition to static parsing, the processing engine executes scripts to recover dynamically constructed links. The leak quantification techniques are evaluated both in a controlled environment, and on real web browsing data to demonstrate their effectiveness in isolating information in outbound web traffic.

1.4.4 Inferring Malicious Activity with a Whitelist

The methods and systems for analyzing web application behavior and information leaks are only tools. They provide valuable feedback on the characteristics of network traffic, but do not constitute a means for tagging such traffic as malicious or benign. After all, most malicious software is no fundamentally different in its behavior than legitimate network applications. The culmination of these network analysis techniques is a whitelist of allowed behavior that separates the good traffic from the bad. All web requests that differ in formatting, timing, or unconstrained bandwidth from a standard web browser are sent to the whitelist for final judgment. The whitelist contains entries that specify the type of trigger (formatting, timing, bandwidth), the associated application, and, optionally, the client(s), server(s), and times for which the entry is valid. Alerts that do not match a known good application on the whitelist are considered malicious and flagged for further investigation.

1.5 Thesis Organization

Chapter 2 surveys related security research. Chapter 3 covers the design and implementation of the Storage Capsule security system. Chapter 4 describes methodology for identifying web applications. Chapter 5 presents algorithms for precisely measuring information content in outbound web traffic. Chapter 6 discusses techniques for inferring malicious activity with the help of a whitelist. Chapter 7 talks about limitations of the security systems presented in this thesis. Finally, Chapter 8 concludes and discusses future work.

المنارة للاستشارات
www.manaraa.com

CHAPTER 2

SURVEY OF RELATED WORK

2.1 Overview

Research that is related to this work can be divided into two main categories: host-based security controls and network-based security systems. Host-based security controls covered in this chapter include information flow control systems, virtual machine-based security systems, and file encryption techniques. Network-based security systems can be subdivided into those that control information flow to prevent security breaches, and those that try to detect suspicious activity from compromised computers. Some intrusion detection systems run on the host, but these systems have more in common with the network-based security systems presented in this thesis, so they are discussed in that context. This survey of related work serves to position the research in this thesis and highlight the state of the art in computer security as it applies to protecting confidential digital information.

2.2 Host-Level Security Controls

2.2.1 Information Flow Control

There has been a great deal of research on controlling the flow of sensitive information within a single computer. This research falls into two categories: inter-process flow control and intra-process flow control. Inter-process flow control is concerned with how programs that access sensitive information interact with one another. The primary example in this area is SELinux, a security module that uses mandatory access control [NSA09] and can stop unwanted information flows. Intra-process mechanisms focus on methods for writing and analyzing programs that one can track sensitive data in individual variables. These tools can check, for example, if a program could possibly leak one person’s password to another user.

المنارة للاستشارات

www.manaraa.com

Mandatory Access Control

Mandatory access control (MAC) involves enforcement of access control policies on high-level objects (typically files, processes, etc.) in a computer system to fulfill security goals. Some of the original MAC policies, such as the Biba integrity model [Biba75] and the Bell-LaPadula secrecy model [Bell75], were fairly simple. They prevented flows of data from LOW integrity to HIGH integrity and from HIGH secrecy to LOW secrecy. They achieved their security goals, but unfortunately prevented most useful applications from working properly. A more modern MAC system that is popular for high-security computers is SELinux [NSA09], a component of the Linux kernel that is supported by the Linux Security Modules framework [Wright02]. SELinux allows different MAC policy models, but an extended Type Enforcement model [Boebert85] is the most widely-used for policy creation. The TE model is much more flexible than both the Biba and Bell-LaPadula models, and is more conducive to practical applications.

One of the main challenges in effectively deploying SELinux is configuring a policy set that both guarantees security and allows applications run correctly. As Jaeger et al. point out [Jaeger03], checking to see if applications can run is much easier than identifying security vulnerabilities. Later work has looked at ways of automatically verifying integrity policies for security-critical applications [Shankar06]. However, there are currently no general techniques for easily specifying mandatory access control policies to secure arbitrary applications. Effective mandatory access control is only practical for systems with well-defined application sets. MAC would have a hard time protecting personal computers that download and install various programs from the Internet. Most computers today do not employ mandatory access control, as they run the Microsoft Windows operating system, which does not yet support MAC.

Static Flow Analysis

One significant limitation of MAC is the coarse granularity with which it enforces access control policies. Tainting an entire process that has touched sensitive information makes it impossible for mandatory access control to secure a program that manipulates data with different confidentiality levels. Researchers have developed program-level flow control tools to provide finer-grained taint tracking and deliver guaranteed security for a wider range of applications.

Denning et al. proposed the first mechanism for statically checking program information flow properties [Denning77]. Their mechanism was more efficient than previous approaches that involved dynamically monitoring information flow. However, their programming model requires

المنارة للاستشارات
www.manaraa.com

extensive manual annotation of variables with security classes, and would not allow many applications to run without violating security constraints.

JFlow [Myers99] is a more modern program-level flow tracking language based on Java. Jif [Myers01] is an implementation of a JFlow. JFlow differs from previous systems in that it focuses on practical security for real programs. JFlow relies on static labeling with the decentralized label model [Myers97] to specify security policies that protect sensitive information within a computer program. These policies dictate the set of entities that may read each piece of data. JFlow statically checks that operations in a program do not leak information in violation of these policies. JFlow also allows declassification, whereby one entity can perform security checks and remove labels from an object.

Even with some support for declassification, flow-tracking mechanisms that taint variables derived from sensitive data are still too restrictive in some cases. Instead of completely blocking flows from high secrecy to low secrecy principals, recent work by McCamant et al. proposes quantitative flow tracking [McCamant08]. Traditional flow control tools mark each piece of data with a sensitivity label. Anything derived from that label is also considered sensitive. This approach tells you if a leak is possible, but not *how much* data can be leaked. McCamant et al. treat information as an "incompressible liquid," measuring the number of sensitive bits that can flow from one variable to another at each operation in a program. The output of their system facilitates fuzzy decision-making about whether to allow or fix information leaks based on their size. This model can help in the design of any security-critical application, even those for which flow control languages like JFlow would be impractical.

Although intra-program flow control systems help ensure that applications behave securely, they suffer from a number of major limitations. First, the source code of the program in question must be analyzed with the static checker, and sometimes requires annotations. This makes static flow-control tools impractical for most legacy programs. Even more importantly, however, static checkers do not help against malicious software or legitimate software that falls under the control of an attacker at runtime. They are not designed to prevent malware from reading sensitive information and leaking it to an external network. The systems presented in this thesis use some of the same principles for analyzing the flow of sensitive information, but are able to operate in an adversarial environment without assumptions about software integrity.

المنارة للاستشارات

www.manaraa.com

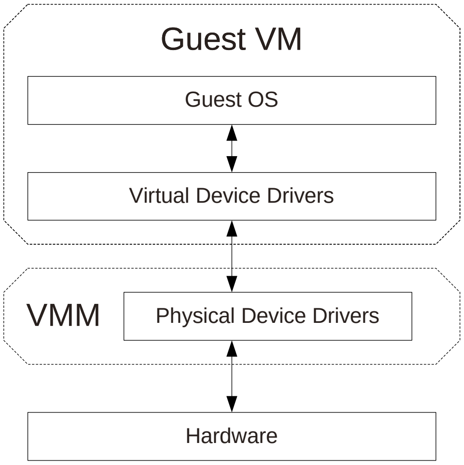


Figure 2.1. Architecture of a virtual machine system. One or more guest virtual machines rely on the virtual machine monitor (VMM) to multiplex hardware resources for each VM. The VMM also provides resource and fault isolation between the virtual machines.

2.2.2 Benefits of Virtualization

Virtualization has become a popular technology for a wide variety of security systems. Figure 2.1 shows the high-level architecture of a virtual machine system. The virtual machine monitor (VMM) runs at the lowest layer and mediates hardware access for each virtual machine (VM). Each VM in turn sees what looks like a dedicated hardware interface, but is actually going through a virtual translation layer to reach the actual hardware. Virtual machine architectures provide a number of benefits and abstractions that are unavailable in a traditional operating system. Some of advantages of virtualization were originally outlined by Goldberg [Goldberg74]. Chen et al. [Chen01] more recently argue for the mass migration of several application classes, some of which involve security, to a virtual machine architecture. These following properties of virtualization make it an attractive technology for building security systems:

Isolation. Multiple virtual machines can run on the same hardware with a high degree of isolation from one another. Each virtual machine has its own view of the disk, memory, and CPU that other virtual machines cannot access. Like separate physical machines, VMs that need to communicate with each other usually do so over a network interface, leading to approximately the same level of isolation experienced by physical machines. The underlying virtual machine monitor is also isolated from virtual machines to a much higher degree than a standard operating system is from applications. This is because the VMM only provides a minimal set of interfaces to multiplex memory and CPU for different virtual machines rather than a full set of system calls.

Robustness. The virtual machine monitor is much smaller than a standard operating system, and thus has fewer parts that can break. The Xen VMM is implemented in under 50,000

www.manaraa.com

lines of code [XenSource09], compared to 5.7 million lines for the Linux 2.6 production kernel, and approximately 40 million for Windows XP [Delio04]. The VMM has only a few interfaces to provide a minimal set of services for virtualization. It is considered much more reliable and secure than a standard OS. Vulnerability reports reinforce this fact, showing only 9 security vulnerabilities for Xen 3.x [Secunia09a], while showing 168 for the Linux 2.6.x Kernel [Secunia09b].

Flexibility. Virtual machines running on the same host can have different operating systems. This enables side-by-side execution of a high-security VM running SELinux [NSA09] and a VM with a commodity OS running a computer game. The security and usability of the entire physical machine is not constrained by the limitations of an individual operating system.

Visibility. Visibility goes hand in hand with isolation. Although security systems running in the kernel of a standard operating system have a high degree of visibility as well, they are not sufficiently isolated from security threats and can be disabled by attack code that is running in the same protection domain. Virtualization allows the VMM or a trusted VM to fully view another virtual machine’s disk, memory, and CPU state from an isolated location.

Control. Because the VMM sits between virtual machines and hardware, it can control access by virtual machines to external devices and other VMs. This level of control for all external communication by an isolated entity is unavailable for a traditional computer. The closest mechanism is a network firewall. The VMM’s ability to control other devices, such as the disk, the display, and human input, makes it a powerful tool for implementing security systems.

Management. One huge advantage of using virtual machines is the ability to easily save, restore, and copy the state of an entire VM. This helps for resetting a VM to clean state if its integrity has been compromised, or for creating a new virtual machine for a specific dangerous task, such as opening a suspicious executable file.

Despite its numerous advantages, virtualization is not without its problems. Garfinkel et al. warn that virtualization is not a completely free lunch, especially when it comes to security [Garfinkel05]. The large number of saved machine states and rapid branching can make patching and configuration much more challenging. Identifying and securing every virtual machine can be an administrative nightmare in an environment with a large number of branches, checkpoints, and different operating systems. The Capsule security system presented in this thesis does not suffer from these limitations because it is designed to have only one primary virtual machine with a fairly straight execution path. The only time that Capsule saves and restores VM state is when it transitions to and from secure editing mode. These transitions are meant to be short term – a user

المنارة للاستشارات
www.manaraa.com

is unlikely to stay in secure editing mode for more than a few hours – and should have minimal impact on patching and management for the primary virtual machine.

2.2.3 Virtualization-Based Security Systems

Logging and Replay

Many of the latest security systems rely on virtual machine technology. One popular application of virtualization is secure logging and replay. ReVirt records long-term virtual machine execution for later analysis [Dunlap02]. Backtracker builds upon the logging capabilities of ReVirt to trace causality during the course of an intrusion and identify events that led up to the current state of objects in a system [King05]. IntroVirt goes a step further and automatically detects past and present exploitation of security holes by actively checking vulnerability predicates [Joshi05]. All of these systems take advantage of the isolation provided by virtual machines to log actions from the VMM where they are safe from attack by a compromised guest operating system. The main purpose of VM-based logging and replay is analyzing intrusions into a system with a greater level of visibility and isolation from attackers than was possible with previous logging systems. In contrast, the work related to virtualization in this thesis does not address the problem of intrusion detection or analysis. Capsule operates under the assumption that the guest VM may be compromised, and seeks to provide confidentiality for sensitive information on a potentially hostile virtual machine.

Malware Analysis

Virtual machines have become tool of choice for analyzing malicious software. Running suspicious programs in a virtual machine and observing their behavior is much easier than using a standard computer. An analyst can simply revert a VM to a previous state if it becomes infected with a virus, while performing the same task without virtualization would entail reloading the entire hard drive. VM-based malware analysis has become so popular that many malware programs will check if they are running in a virtualized environment using a technique such as Red Pill [Rutkowska05], and behave differently if they are. The Cobra malware analysis framework provides a solution for efficiently examining malware in a manner that the program under analysis cannot detect [Vasudevan06]. In addition to analyzing malware binaries, virtual machines are also an effective means for deploying honeypot systems. Provos first introduced the idea of a wide-scale “Virtual Honeypot Framework” that involved a single machine masquerading as many computers to elicit network attacks [Provos04]. Although Provos did not use virtual machines, later work on the Potemkin Virtual Honeyfarm shows how to quickly and

المنارة للاستشارات
www.manaraa.com

reliably spawn large numbers of virtual machines on one computer for the purpose of analyzing attacks [Vrable05]. The work in this thesis is similar in that it assumes potential malware infection on the primary virtual machine, but it does not try to analyze or characterize malicious activity.

Trusted Computing

The Terra trusted computing platform [Garfinkel03a] exploits the flexibility of a virtual machine architecture to run trusted code side-by-side with standard low-integrity code in a separate virtual machine. Terra is based on the chain-of-trust principle for verifying critical system components, similar to the approach taken by trusted boot [Sailer04]. During the boot process, each successive system component verifies the cryptographic checksum of the next piece of code with the help of a trusted platform module (TPM) [TCG06] for performing secure cryptographic operations. This ensures system integrity at startup. Terra departs from trusted boot when the virtual machine monitor gains control. Instead of verifying all guest virtual machines, Terra only checks VMs that are considered high-integrity, while allowing low-security VMs to run untrusted code.

Terra’s approach is similar to Storage Capsules in that it enables high-security functionality on a computer that runs low-security code. One could configure Terra to have a trusted virtual machine for editing encrypted sensitive documents that is verified at boot time. Given that the software on this VM is free from security bugs, it would allow the user to create, edit, and transfer some documents with guaranteed confidentiality. The problem comes when this trusted VM wants to access low-integrity data. If the user creates a new file with the trusted editing VM, then there are no issues. However, if a document includes information from external sources, such as the Internet or e-mail, this data could compromise the trusted VM’s integrity and cause it to leak information. This restriction makes an integrity-preserving approach using a trusted VM computing platform less usable than Storage Capsules, which support editing from a low-integrity system.

Introspection and Intrusion Detection

The isolation provided by virtual machines lends itself well to enhancing the security of host-based intrusion detection systems (HIDS). Traditionally, HIDS had the advantage of greater visibility into actions on the host, but were vulnerable to attack because they ran in the same protection domain as malicious software. Garfinkel et al. recognized this shortcoming and proposed a new architecture for HIDS that uses *virtual machine introspection* (VMI)

المنارة للاستشارات
www.manaraa.com

[garfinkel03c]. With VMI, the intrusion detection system runs beneath the virtual machine under examination, and inspects its state to detect suspicious behavior. The catch is that a VMI approach must reconstruct higher-level semantics from raw binary data using information about the operating system. This is because high-level representations of objects such as files, processes, and threads are unavailable from the perspective of the virtual machine monitor.

Capsule does not use virtual machine introspection to derive high-level information about the state of low-integrity virtual machines. Instead, it uses a message-passing interface to communicate with the primary VM, which operates under the assumption that any message received from the VM is not trustworthy. Capsule's other functionality that affects the primary virtual machine, such as saving and restoring system state, does not require interpretation of high-level objects in the VM.

2.2.4 Host-Level File Encryption

There are a number of security products available for encrypting and protecting files on a local computer. Some compression utilities come with archive encryption capabilities [Roshal09, Winzip09]. Another program called TrueCrypt can encrypt individual file stores, or entire volumes [TrueCrypt09]. The advantage of TrueCrypt over archival compression utilities is that it has better performance for random access. Microsoft Windows Vista also comes with full-drive encryption technology known as BitLocker [Microsoft09]. These systems are descendents of the original cryptographic file system for UNIX, which was the first to address performance and security limitations of encryption utilities by moving encryption into the file system layer [Blaze93].

File encryption is designed to protect against a few different threats. First, it allows one to send a sensitive document or archive over an untrusted medium, such as the Internet. If the file is encrypted, then only the intended recipient(s) who have the decryption key can view the data. The primary purpose of whole-drive encryption is to protect data if the disk drive is stolen or lost. Unfortunately, file encryption systems cannot safeguard sensitive information while it is decrypted on the end host. The data must exist in plain text while the user is editing or viewing documents. This is generally considered to be the weakest link for cryptographic file systems. Their ability to protect confidential files from malicious software on the end host is very limited. Capsule, on the other hand, uses file encryption but protects sensitive plain-text information from eavesdropping on the end host.

المنارة للاستشارات
www.manaraa.com

2.3 Network-Level Security Controls

2.3.1 Threat Prevention

One class of network security systems helps protect both end host integrity and confidentiality by preemptively blocking undesirable communication. Preventing two specific entities from interacting with one another helps network threat prevention systems stop a large portion of incoming attacks, as well as outbound information leaks. These security mechanisms generally fall into two categories based on whether or not they examine application-layer data in network connections. Network flow-control tools that only look at data below the application layer include firewalls and content filters. Intrusion prevention systems (IPSs) and data loss prevention (DLP) systems examine high-level information at the application layer. We also discuss anti-virus software here, as it is shares many similarities with intrusion prevention systems.

Security threat prevention is much better than passive detection when it is successful. However, threat prevention systems are limited by the need to have an extremely low false positive rate. Improperly blocking legitimate network traffic can be detrimental to productivity in an organization. This forces prevention systems to be conservative in their traffic filtering, which limits their ability to detect a wide range of attacks. Furthermore, having a prevention system in place allows attackers to get active feedback about what is being blocked, and modify their behavior to bypass security controls. Despite these limitations, threat prevention systems such as firewalls serve as an effective first line of defense, immediately blocking a large number of network attacks. The remainder of this section describes the operation of several network-based threat prevention systems in greater detail.

Firewalls

Network firewalls govern the flow of packets in and out of computing systems. Standard firewalls match packets based on their transport-layer protocol, source/destination addresses, and source/destination port numbers. In this way, firewalls can protect vulnerable services from remote exploitation by blocking the ports on which they receive packets. Firewalls can also block outbound traffic on specific ports. This configuration is known as egress filtering. It can help prevent unwanted applications from accessing the Internet. Coarse-grained filtering by firewalls is effective because it completely blocks traffic that is disallowed by the firewall policy. Unfortunately, firewalls provide zero protection for traffic that they allow. Firewalls do not help secure services that must be accessed from outside the network. It is also easy for applications to

المنارة للاستشارات
www.manaraa.com

bypass egress filters by switching their communications to an allowed port for another program. Firewalls are effective at filtering out a large portion of unwanted traffic, but can be easily circumvented.

Content Filters

Content filters refer to the set of network flow control tools that prevent computers from accessing undesirable web servers [OpenDNS09, Websense09]. Content filters focus more on the destination server than the actual content of network traffic, but got their name because their original purpose was blocking inappropriate web content. Modern content filters can be set up to block many types of unwanted websites, or even allow traffic to only a limited set of trusted sites. The goal of content filters is often to restrict users from accessing websites that impede their productivity or threaten their security. However, content filters can also serve to protect confidentiality by cutting off avenues for information leakage. They may block servers that collect stolen data from spyware programs, or stop malicious insiders from sending out confidential documents through web mail. One drawback of content filters is that they may limit the productivity of users who need access to a broad range of sites for their work.

Though they provide some security benefits, content filters are not able to stop sophisticated adversaries from leaking information. This is because an attacker can transmit data through any web server that accepts a post and will display that information to other computers. This includes any site with a message board, forum, wiki, or other collaborative interface. Identifying and blocking all such websites would be extremely. Sites that accept posts represent a large portion of the web, including many useful websites like www.wikipedia.org. Even if current content filters were able to identify all sites that accept posts, blocking them would have a severe impact on usability. Content filters are an inadequate tool for controlling the flow of confidential information.

Intrusion Prevention Systems

The goal of intrusion prevention systems (IPSs) [IBM09b, TippingPoint09] is to identify and block malicious network traffic. This includes attacks from worms, bots, and hackers. IPSs will use a *blacklist* of known bad traffic to detect attack traffic. IPSs are very similar to some intrusion detection systems, which also use a blacklist to detect bad traffic but do not actively block malicious activity. From an information flow-control perspective, IPSs only serve to prevent the flow of malicious data into a network; they do not stop information leaks. Like all systems that rely on a blacklist, they are also unable to block new and unknown attacks.

المنارة للاستشارات
www.manaraa.com

Anti-Virus Software

Although anti-virus (AV) software [McAfee09, Symantec09] typically runs on the end host, the way that it detects and blocks malicious software is very similar to a network intrusion prevention system. Some AV software actually comes with an active protection component that scans incoming network traffic for attacks, similar to an intrusion prevention system. The main limitation of anti-virus programs is that they rely on a blacklist to detect malware. This prevents them from detecting new malware payloads, which are very easy to build from the perspective of an attacker. One can create malware with a new signature by simply changing a few unimportant pieces of functionality and recompiling the program. Furthermore, hackers have access to anti-virus signatures and can test to make sure that their new malware goes undetected. Anti-virus software is quite effective against viruses and other malicious programs that have a wide distribution, but does not provide protection against more sophisticated attacks.

Data Loss Prevention Systems

The purpose of data loss prevention (DLP) systems [RSA07, Vontu09] is to inspect outgoing network traffic for the presence confidential information. If a DLP system detects sensitive data, it can the block the stream or log an alert for further auditing. Like IPSs, data loss prevention systems look at the payloads of network connections to decide whether they should be blocked. Unfortunately, this only consists of checking for known confidential information, which prevents DLP systems from being able to detect encrypted or obfuscated information leaks. The principles of data hiding and steganography show that it is nearly impossible, in the general case, to determine with certainty whether some data contains a hidden message [Petitcolas99]. In practice, DLP systems are helpful for stopping accidental leaks, but have no way of protecting against a malicious insider who understands encryption or data hiding.

2.3.2 Intrusion Detection

A large portion of security research focuses on the problem of detecting malicious activity by looking at network traffic or host-level actions. Systems that process this activity and identify suspicious behavior are known as intrusion detection systems (IDSs). There are two families of intrusion detection systems: those that rely on signatures, and those that detect behavioral anomalies. Signature-based IDSs benefit from a low false positive rate. It is also easy for an analyst to respond to alerts because they are associated with known attacks. Unfortunately, signature-based IDSs cannot detect new or unknown threats, making them ineffective against most targeted attacks. Behavioral anomaly detection systems are better at responding to unknown

المنارة للاستشارات
www.manaraa.com

threats because they use models to characterize suspicious activity. However, anomaly detection systems are still vulnerable to evasion by attackers that mimic legitimate activity. They are also more prone to false positives than signature-based systems because they look for suspicious patterns rather than precisely known attacks. This makes it harder for administrators to respond to alerts as well, because less information is known to a behavioral anomaly detection system about the nature of attacks.

The network traffic analysis techniques in this thesis fall into the category of anomaly-based intrusion detection systems. They differ from previous approaches, however, due to the use of a whitelist to filter out alerts and reduce false positives. In this way, there are similarities with systems that use a blacklist of signatures to identify attacks. This section describes a number of intrusion detection systems in detail, and compares them to the systems presented in this thesis.

Network-Based Signature Detection

The most common type of system that people think of when they hear intrusion detection is a network-based signature detection system. Classic signature-based NIDSs have a repository of signatures for known network attacks. They check traffic in real time against these attack signatures, and raise alerts for matching streams. An administrator then views and responds to the alerts, which may indicate that a system has been compromised by a hacker. Examples of traditional signature-based NIDSs include Snort [Roesch99], Bro [Paxson98], and ISS RealSecure [IBM09a]. Some signature-based NIDSs, such as Peakflow X [Arbor09], look for malware attack patterns based on sequences of network connections rather than string matching on individual packets. These systems have the benefits mentioned earlier of straightforward attack remediation and low false positive rates.

Some signature-based NIDSs, however, are vulnerable to evasion by changing attack patterns, as demonstrated by Vigna et al. [Vigna04]. More recent research tries to address this limitation by creating vulnerability-specific signatures that are impossible to evade (while still exploiting the same vulnerability) [Brumley06, Newsome05b]. Even if foolproof signatures are in place, NIDSs are vulnerable to new and unknown attacks. This includes attacks by malicious insiders, which are a serious threat and generally do not have a signature. As the volume of malicious software and security vulnerabilities continues to increase, even keeping up with signatures for publicly known attacks is a difficult task. Signature-based NIDSs simply do not provide complete protection against network security threats.

المنارة للاستشارات
www.manaraa.com

Network-Based Anomaly Detection

Many researchers have made the observation that malicious network traffic exhibits different patterns than legitimate network traffic. Several systems attempt to characterize malicious traffic of different types. Originally, network-based anomaly detection (NBAD) systems primarily used connection statistics to discover suspicious communication patterns between computers [Mukherjee94]. This approach was successful at detecting many attacks without the help of signatures. However, since the creation of NBAD systems, the model for network interaction has changed significantly. Now, most client computers block all incoming traffic with a firewall by default, only allowing client/server communication. This has forced hackers to adopt the client/server model for exploiting computers as well. Modern network attacks are often delivered through e-mail or a malicious website rather than direct inbound connections. Traditional statistical anomaly detection techniques that look at low-level connection information have a difficult time detecting these types of threats because they are hard to differentiate from legitimate e-mail or web browsing.

More modern NBAD systems focus on network traffic payloads. Kruegel et al. demonstrate how to detect anomalies by measuring service-specific payload statistics (e.g. specific to web browsing, e-mail, etc.) related to the type of request, request length, and payload byte distribution [Kruegel02]. Wang et al. improve upon these techniques by using a more fine-grained byte distribution analysis [Wang04]. These systems were successful at detecting a large portion of new network attacks from the DARPA intrusion detection data set. However, NBAD systems that focus on payload statistics are inherently vulnerable to mimicry attacks. If a hacker knows about payload statistics that are in use for anomaly detection, he or she can adjust malicious traffic to fit within the set of normal activity. Another shortcoming of statistical payload analysis is that it does not catch attacks that occur by way of legitimate protocol usage. For example, if spyware uploads sensitive passwords to a rogue web server using normal HTTP requests, this traffic will look exactly the same, from a statistical perspective, as legitimate log-in requests.

Some NBAD systems look for executable shell code in traffic payloads [Polychronakis06]. The presence of shell code indicates a buffer overflow exploitation attempt. These systems can effectively block such attacks. However, buffer overflow exploits represent only a small portion of unwanted network traffic. Many attacks involve tricking users into downloading full programs or plug-ins that compromise their computers rather than exploiting a buffer overflow. Furthermore, once a machine has become infected, it is unlikely to generate buffer overflow payloads, unless it is actively attacking another computer. Spyware or malicious

المنارة للاستشارات
www.manaraa.com

insiders that try to leak sensitive information will not be detected by an NBAD system that looks for executable shell code.

One more recent NBAD system, BotMiner, departs from payload analysis and instead detects malware by correlating network behavior [Gu08]. BotMiner's goal is to detect a specific type of malicious software, known as a bot, which controls a computer to do the bidding of central bot master, who has compromised numerous machines with the bot software. BotMiner takes advantage of the fact that more than one computer will be infected with the same bot in most cases, and these computers will exhibit the same malicious behavior. While certain activities, such as scanning or communicating over internet relay chat (IRC) with an unknown server, are uncommon but not malicious by themselves, the occurrence of such behavior from multiple computers in the same network is highly indicative of a bot compromise. Unfortunately, BotMiner cannot detect spyware that only sends confidential information to a rogue web server, as its requests do not constitute anomalous behavior that can be correlated across multiple infected machines. BotMiner is also limited by the fact that it can only identify malicious activity if more than one computer is compromised. For targeted attacks, a hacker could avoid correlation-based detection by only infecting one computer with a particular type of malware.

Host-Based Anomaly Detection

Although they operate on an individual computer, host-based intrusion detection systems (HIDS) share some commonalities with the network threat detection systems in this thesis. The main difference between HIDS and NIDS is that HIDS have access to a myriad of local information that cannot be seen over the network. They can view anything at all on a computer, including program execution, system calls, and device I/O. System calls are a particularly popular target of analysis, as they are the bridge between compromised applications and the underlying system. Hofmeyr et al. describe an intrusion detection system that discovers malicious behavior by looking for sequences of system calls that were previously unseen for a particular application [Hofmeyr98]. The insight behind their approach is that programs tend to behave differently when they have become compromised. One limitation of system call HIDS is that they require extensive calibration to determine the set of allowable system call behavior. Performing such calibration would be impractical for personal computers that frequently install new software. Furthermore, training data must be collected during a period in which the program in behaving legitimately. System call HIDS cannot protect against the installation of programs packaged with Trojan horse malware because they would behave maliciously during the training. System call HIDS can detect a wide variety of attacks, but by no means provide complete host-level

المنارة للاستشارات
www.manaraa.com

protection. They serve as an important influence for the work in this thesis, however, because system call HIDS are an early example of using whitelists for intrusion detection.

Tripwire is an early host-based IDS that looks for modifications to critical system files as an indicator of malicious behavior [Kim94]. Tripwire is very similar, in fact, to work on trusted computing that checks the integrity of all critical files at boot time. Tripwire will sit and wait for changes to important files, and then notify a system administrator when they occur. This can help detect the installation of persistent root kit programs, or modifications to the system configuration, such as changing the password file, that grant the attacker permanent access to the target machine. Tripwire is only useful against some attacks. If a hacker infects a machine and uses it for nefarious network activity without modifying any files, then it would be impossible for Tripwire to detect the compromise. Tripwire is also vulnerable to malicious software with elevated privileges that modifies the file system at a layer below Tripwire's detection hooks. However, this vulnerability could be fixed by updating tripwire to use virtual machine technology and run below the virtual machine in question. Tripwire is largely orthogonal to work in this thesis because it only examines events that indicate a change in system configuration on the local host, rather than looking for breaches of confidentiality.

المنارة للاستشارات
www.manaraa.com

CHAPTER 3

PROTECTING CONFIDENTIAL DATA ON PERSONAL COMPUTERS WITH STORAGE CAPSULES

3.1 Introduction

Traditional methods for protecting confidential information rely on upholding system integrity. If a computer is safe from hackers and malicious software (malware), then so is its data. Ensuring integrity in today's interconnected world, however, is exceedingly difficult. Trusted computing platforms such as Terra [Garfinkel03a] and trusted boot [Sailer04] try to provide this integrity by verifying software. Unfortunately, these platforms are rarely deployed in practice and most software continues to be unverified. More widely-applicable security tools, such as firewalls, intrusion detection systems, and anti-virus software, have been unable to combat malware, with 100 to 150 million infected machines running on the Internet today according to a recent estimate [Weber07]. Security mechanisms for personal computers simply cannot rely on keeping high integrity. Storage Capsules address the need for access to confidential data from compromised personal computers.

Some existing solutions for preserving confidentiality do not rely on high integrity. One example is mandatory access control (MAC) with multi-level security (MLS) and policies that abide by the Bell-LaPadula secrecy model [Bell75]. With these policies, MAC can prevent entities that read confidential information from communicating over the network. This policy set achieves the goal of preventing leaks in the presence of malware. However, defining policies that are both correct and allow normal programs to run can be difficult. Even if programs can run, their utility may be severely limited. For example, documents saved by a word processor that has ever read secret data could not be sent as e-mail attachments. Another embodiment of the same principle can be seen in an "air gap" separated network where computers are physically disconnected from the outside world. Unplugging a compromised computer from the Internet will stop it from leaking information, but doing so greatly limits its utility. Both mandatory access

المنارة للاستشارات
www.manaraa.com

control with strict outbound flow policies and air gap networks are rarely used outside of protecting classified information due to their severe impact on usability.

This chapter introduces Storage Capsules, a new mechanism for protecting sensitive information on a local computer. The goal of Storage Capsules is to deliver a comparable level of security as a mandatory access control system, but for standard applications running on a commodity operating system. Storage Capsules meet this requirement by enforcing policies at a system-wide level using virtual machines. The user's system can also downgrade from high-secrecy to low-secrecy by reverting to a prior state using virtual machine snapshots. Finally, the system can obtain updated Storage Capsules from a declassification component, which makes sensitive data available at a lower secrecy level by encrypting it with a secret key.

Storage Capsules are analogous to encrypted file containers from the user's perspective. When the user opens a Storage Capsule, a snapshot is taken of the current system state and device output is disabled. At this point, the system is considered to be in *secure mode*. When the user is finished editing files in a Storage Capsule, the system is reverted to its original state – discarding all changes except those made to the Storage Capsule – and device output is re-enabled. The storage capsule is finally re-encrypted by a trusted component.

Storage Capsules guarantee protection against a compromised operating system or applications. Sensitive files are safe when they are encrypted *and* when being accessed by the user in plain text. The Capsule system prevents the OS from leaking information by erasing the OS's entire state after it sees sensitive data. It also stops covert communication by fixing the Storage Capsule size and completely re-encrypting the cipher-text every time it is accessed by the OS outside of secure mode. Our threat model assumes that the primary operating system can do anything at all to undermine the system. The threat model also assumes that the user, hardware, the virtual machine monitor (VMM), and an isolated secure virtual machine are trustworthy. The Capsule system protects against covert channels in the primary OS and Storage Capsules, as well as many (though not all) covert channels at lower layers (disk, CPU, etc.). One of the contributions of this thesis is identifying and suggesting mitigation strategies for numerous covert channels that could potentially leak data from a high-secrecy VM to a low-secrecy VM that runs after it has terminated.

We evaluated the impact that Storage Capsules have on the user's workflow by measuring the latency of security level transitions and system performance during secure mode. We found that for a primary operating system with 512 MB of RAM, transitions to secure mode took about 4.5 seconds, while transitions out of secure mode took approximately 20 seconds. We also compared the performance of the Apache build benchmark in secure mode to that of a native

www.manaraa.com

machine, a plain virtual machine, and a virtual machine running an encryption utility. Overall, Storage Capsules added 38% overhead compared to a native machine, and only 5% compared to a VM with encryption software. The typical workload for a Storage Capsule is expected to be much lighter than an Apache build. In many cases, it will add only a negligible overhead.

The remainder of this chapter is laid out as follows. Section 3.2 gives an overview of the usage model, the threat model, and design alternatives. Section 3.3 outlines the system architecture. Section 3.4 describes the operation of Storage Capsules. Section 3.5 examines the effect of covert channels on Storage Capsules. Section 3.6 presents evaluation results. Finally, section 3.7 concludes and discusses future work.

3.2 Overview

3.2.1 Storage Capsules from a User's Perspective

From the user's perspective, Storage Capsules are analogous to encrypted file containers provided by a program like TrueCrypt [TrueCrypt09]. Basing the Capsule system off of an existing and popular program's usage model makes it easier to gain acceptance. The primary difference between Storage Capsules and traditional encryption software is that the system enters a secure mode before opening the Storage Capsule's contents. In this secure mode, network output is disabled and any changes that the user makes outside of the Storage Capsule will be lost. The user may still edit the Storage Capsule contents with his or her standard applications. When the user closes the Storage Capsule and exits secure mode, the system reverts to the state it was in before accessing sensitive data.

One motivating example for Storage Capsules is protecting financial information. A person, call him Bob, might have a spreadsheet that contains bank account numbers, his social security number, tax information, etc. Bob wants to be able to download monthly statements from his bank and copy information into the spreadsheet. However, Bob is worried about spyware stealing the data because it could lead to identity theft. Being a diligent computer user, Bob stores the spreadsheet in an encrypted file container. Every month, Bob downloads his financial statement, opens up the container, and updates the spreadsheet. Unfortunately, Bob is still completely vulnerable to spyware when he enters the decryption password and edits the spreadsheet. Storage Capsules support the same usage model as normal encrypted file containers, but also deliver protection against spyware while the user is accessing sensitive data.

Storage Capsules have some limitations compared to encrypted file containers. These limitations are necessary to gain additional security. First, changes that the user makes outside of the encrypted Storage Capsule while it is open will not persist. This benefits security and privacy

المنارة للاستشارات

www.manaraa.com

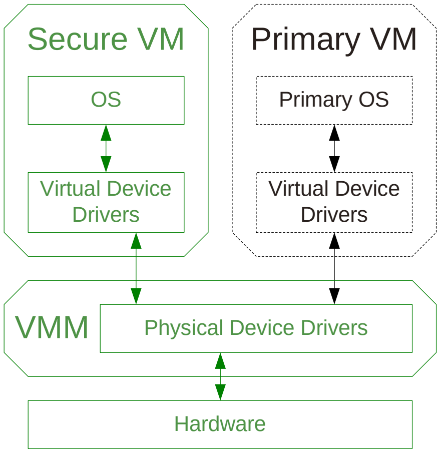


Figure 3.1. In the Storage Capsule system architecture, the user's primary operating system runs in a virtual machine. The secure VM handles encryption and declassification. The dotted black line surrounding the primary VM indicates that it is *not* trusted. The other system components are trusted.

by eliminating all traces of activity while the container was open. Storage Capsules guarantee that the OS does not inadvertently hold information about sensitive files, such as described by Czeskis et al. for the case of TrueCrypt [Czeskis08]. Unfortunately, any work from computational or network processes that may be running in the background will be lost. One way to remove this limitation would be to fork the primary virtual machine and allow a copy of it to run in the background. Allowing low- and high-secrecy VMs to run at the same time, however, reduces security by opening up the door for a variety of covert channels.

3.2.2 Threat Model

Storage Capsules are designed to allow a compromised operating system to safely edit confidential information. However, some trusted components are necessary to provide security. Figure 3.1 shows the architecture of the Capsule system, with trusted components having solid lines and untrusted components having dotted lines. The user's primary operating system runs inside of a *primary VM*. The applications, the drivers, and the operating system are not trusted in the primary VM; it can behave in any arbitrary manner. A *virtual machine monitor (VMM)* runs beneath the primary VM, and is responsible for mediating access to physical devices. The VMM is considered part of the trusted computing base (TCB). The Capsule system also relies on a *Secure VM* to save changes and re-encrypt Storage Capsules. This secure VM has only a minimal set of applications to service Storage Capsule requests, and has all other services blocked off with a firewall. The secure VM is also part of the TCB.

The user is also considered trustworthy in his or her intent. Presumably, the user has a password to decrypt each Storage Capsule and could do so using rogue software without going

المنارة للاستشارات
www.manaraa.com

into secure mode and leak sensitive data. The user does not require full access to any trusted components, however. The main user interface is the primary VM, and the user should only interact with the Secure VM or VMM briefly using a limited UI. This prevents the user from inadvertently compromising a trusted component with bad input.

The threat model assumes that malicious software may try to communicate covertly within the primary VM. Storage Capsules are designed to prevent a compromised primary OS from saving data anywhere that will persist through a snapshot restoration. However, Storage Capsules do not guarantee that a malicious primary VM cannot store data somewhere in a trusted component, such as hardware or the VMM, in such a way that it can recover information after leaving secure mode. We discuss several of these covert channels in more depth later in the chapter.

3.2.3 Designs that do not Satisfy Storage Capsule Goals

The first system design that would not meet the security goals laid out in our threat model is conventional file encryption software [Blaze93, Fruhwirth09, Microsoft09, TrueCrypt09]. Any information stored in an encrypted file would be safe from malicious software, or even a compromised operating system, while it is encrypted. However, as soon as the user decrypts a file, the operating system can do whatever it wants with the decrypted data.

The Terra system [Garfinkel03a] provides multiple security levels for virtual machines using trusted computing technology. Terra verifies each system component at startup using a trusted platform model (TPM) [TCG06], similar to trusted boot [Sailer04]. However, Terra allows unverified code to run in low-security virtual machines. One could imagine a configuration of Terra in which the user's primary OS runs inside of a low-integrity machine, just like in the Capsule system. The user could have a separate secure VM for decrypting, editing, and encrypting files. Assuming that the secure VM always has high integrity, this approach would provide comparable security and usability benefits to Storage Capsules. However, Terra only ensures a secure VM's integrity at startup; it does not protect running software from exploitation. If this secure VM ever loads an encrypted file from an untrusted location, it is exposed to attack. All sources of sensitive data (e-mail contacts, web servers, etc.) would have to be verified and added to the trusted computing base (TCB), bloating its size and impacting both management overhead and security. Furthermore, the user would be unable to safely include data from untrusted sources, such as the Internet, in sensitive files. The Capsule system imposes no such headaches; it can include low-integrity data in protected files, and only requires trust in local system components to guarantee confidentiality.

www.manaraa.com

Another design that would not meet the goals of Storage Capsules is the NetTop architecture [Meushaw00]. With NetTop, a user has virtual machines with multiple security levels. One is for accessing high-secrecy information, and another for low-secrecy information, which may be connected to the Internet. Depending on how policies are defined, NetTop either suffers from usability limitations or would have security problems. First assume that the high-secrecy VM must be able to read data from the low-secrecy VM to load files from external locations that are not part of the trusted computing base. Now, if the high-secrecy VM is prevented from writing anything back to the low-secrecy VM, then confidentiality is maintained. However, this prevents the user from making changes to a sensitive document, encrypting it, then sending it back out over a low-secrecy medium. This effectively makes everything read-only from the high-secrecy VM to the low-secrecy VM. The other alternative – letting the high-secrecy VM encrypt and de-classify data – opens up a major security hole. Data that comes from the low-secrecy VM also might be malicious in nature. If the high-secrecy VM reads that information, its integrity – and the integrity of its encryption operations – may be compromised.

3.3 System Architecture

The Capsule system has two primary modes of operation: *normal mode* and *secure mode*. In normal mode, the computer behaves the same as it would without the Capsule system. The primary operating system has access to all devices and can communicate freely over the network. In secure mode, the primary OS is blocked from sending output to the external network or to devices that can store data. Furthermore, the primary operating system's state is saved prior to entering secure mode, and then restored when transitioning back to normal mode. This prevents malicious software running on the primary OS from leaking data from secure mode to normal mode.

The Capsule system utilizes virtual machine technology to isolate the primary OS in secure mode. Virtual machines also make it easy to save and restore system state when transitioning to or from secure mode. Figure 3.1 illustrates the architecture of the Capsule system. The first virtual machine, labeled Primary VM, contains the primary operating system. This VM is the equivalent of the user's original computer. It contains all of the user's applications, settings, and documents. This virtual machine may be infected with malicious software and is not considered trustworthy. The other virtual machine, labeled Secure VM, is responsible for managing access to Storage Capsules. The secure VM is trusted. The final component of the Capsule system shown in Figure 3.1 is the Virtual Machine Monitor (VMM). The VMM is

www.manaraa.com

responsible for translating each virtual device I/O request into a physical device request, and for governing virtual networks. As such, it can also block device I/O from virtual machines. The VMM has the power to start, stop, save, and restore entire virtual machines. Because it has full control of the computer, the VMM is part of the trusted computing base.

The Capsule system adds three components to the above architecture to facilitate secure access to Storage Capsules. The first is the *Capsule VMM module*, which runs as service inside of the VMM. The Capsule VMM module performs the following basic functions:

- Saves and restores snapshots of the primary VM
- Enables and disables device access by the primary VM
- Catches key escape sequences from the user
- Switches the UI between the primary VM and the secure VM

The Capsule VMM module executes operations as requested by the second component, the *Capsule server*, which runs inside of the secure VM. The Capsule server manages transitions between normal mode and secure mode. During secure mode, it also acts as a disk server, handling block-level read and write requests from the *Capsule viewer*, which runs in the primary VM. The Capsule server has dedicated interfaces for communicating with the Capsule viewer and with the Capsule VMM module. These interfaces are attached to separate virtual networks so that the viewer and VMM module cannot impersonate or communicate directly with each other.

The third component, the Capsule viewer, is an application that accesses Storage Capsules on the primary VM. When the user first loads or creates a new Storage Capsule, the viewer will import the file by sending it to the Capsule server. The user can subsequently open the Storage Capsule, at which point the viewer will ask the Capsule server to transition the system to secure mode. During secure mode, the viewer presents the contents of the Storage Capsule to the user as a new mounted partition. Block-level read and write requests made by the file system are forwarded by the viewer to the Capsule server, which encrypts and saves changes to the Storage Capsule. Finally, the Capsule viewer can retrieve the encrypted Storage Capsule by requesting an export from the Capsule server. The Capsule viewer is not trusted and may cause a denial-of-service at any time. However, the Capsule system is designed to prevent even a compromised viewer from leaking data from secure mode to normal mode.

المنارة للاستشارات
www.manaraa.com

3.4 Storage Capsule Operation

3.4.1 Storage Capsule File Format

A Storage Capsule is actually an encrypted partition that is mounted during secure mode. The Storage Capsule model is based on TrueCrypt [TrueCrypt09] – a popular encrypted storage program. Like TrueCrypt, each new Storage Capsule is created with a fixed size. Storage Capsules employ XTS-AES – the same encryption scheme as TrueCrypt – which is the IEEE standard for data encryption [IEEE08]. In our implementation, the encryption key for each file is created by taking the SHA-512 hash of a user-supplied password. In a production system, it would be beneficial to employ other methods, such as hashing the password many times and adding a salt, to make attacks more difficult. The key could also come from a biometric reader (fingerprint reader, retina scanner, etc.), or be stored on a key storage device like a smart card. Storage Capsules operation does not depend on a particular key source.

With XTS-AES, a different tweak value is used during encryption for each data unit. A data unit can be one or more AES blocks. The Storage Capsule implementation uses a single AES block for each data unit. In accordance with the IEEE 1619 standard [IEEE08], Storage Capsules use a random 128-bit starting tweak value that is incremented for each data unit. This starting tweak value is needed for decryption, so it is stored at the beginning of the file. Because knowledge of the tweak value does not weaken the encryption [Liskov02], it is stored in the clear.

3.4.2 Creating and Importing a Storage Capsule

The first step in securing data is creating a new Storage Capsule. The following tasks take place during the creation process:

1. The Capsule viewer solicits a Storage Capsule file name and size from the user.
2. The viewer makes a request to the Capsule server on the secure VM to create a new Storage Capsule.
3. The viewer asks the user to enter the secure key escape sequence that will be caught by a keyboard filter driver in the VMM. This deters spoofing by a compromised primary VM.
4. After receiving the escape sequence, the VMM module will give the secure VM control of the user interface.
    a. If the escape sequence is received unexpectedly (i.e. when the VMM has not received a request to wait for an escape sequence from the Capsule server), then the VMM module will still give control of the UI to the secure VM, but the secure VM will

المنارة للاستشارات
www.manaraa.com

display a warning message saying that the user is *not* at a secure password entry page.

5. The Capsule server will ask the user to select a password, choose a random starting tweak value for encryption, and then format the encapsulated partition.
6. The Capsule server asks the VMM module to switch UI focus back to the primary VM.
7. After the creation process is complete, the Capsule server will send the viewer a file ID that it can store locally to link to the Storage Capsule on the server.

Loading a Storage Capsule from an external location requires fewer steps than creating a new Storage Capsule. If the viewer opens a Storage Capsule file that has been created elsewhere, it imports the file by sending it to the Capsule server. In exchange, the Capsule server sends the viewer a file ID that it can use as a link to the newly imported Storage Capsule. After a Storage Capsule has been loaded, the link on the primary VM looks the same regardless of whether the Capsule was created locally or imported from an external location.

3.4.3 Opening a Storage Capsule in Secure Mode

At this point, one or more Storage Capsules reside on the Capsule server, and have links to them on the primary VM. When the user opens a link with the Capsule viewer, it will begin the transition to secure mode, which consists of the following steps:

1. The Capsule viewer sends the Capsule server a message saying that the user wants to open a Storage Capsule, which includes the file ID from the link in the primary VM.
2. The Capsule viewer asks the user to enter the escape sequence that will be caught by the VMM module.
3. The VMM module receives the escape sequence and switches the UI focus to the secure VM. This prevents malware on the primary VM from spoofing a transition and stealing the file password.
    a. If the escape sequence is received unexpectedly, the secure VM still receives UI focus, but displays a warning message stating the system is *not* in secure mode.
4. The VMM module begins saving a snapshot of the primary VM in the background. Execution continues, but memory and disk data is copied to the snapshot file if it is written.
5. The VMM module disables network and other device output.
6. The Capsule server obtains the file password from the user.
7. The VMM module returns UI focus to the primary VM.
8. The Capsule server tells the viewer that the transition is complete and begins servicing disk I/O requests to the Storage Capsule.

www.manaraa.com

9. The Capsule viewer mounts a local partition that uses the Capsule server for back-end disk block storage.

The above process ensures that the primary VM gains access to the Storage Capsule contents only after its initial state has been saved and the VMM has blocked device output. The exact set of devices blocked during secure mode is discussed more in the section on covert channels.

Depending on the source of the Storage Capsule encryption key, step 6 could be eliminated entirely. If the key was obtained from a smart card or other device, then the primary VM would retain UI focus throughout the entire transition, except in the case of an unexpected escape sequence from the user. In this case, the secure VM must always take over the screen and warn the user that he or she is not in secure mode.

3.4.4 Storage Capsule Access in Secure Mode

When the Capsule system is running in secure mode, all reads and writes to the Storage Capsule are sent to the Capsule server. The server will encrypt and decrypt the data for each request as it is received, without performing any caching itself. The Capsule server instead relies on the caches within the primary VM and its own operating system to minimize unnecessary encryption and disk I/O. The disk cache in the primary VM sits above the driver that sends requests through to the Capsule server. On the secure VM, disk read and disk write requests from the Capsule server go through the local file system cache before they are sent to the disk. Later, we measure Storage Capsule disk performance during secure mode and demonstrate that is comparable to current encryption and virtualization software.

During secure mode, the VMM stores all writes to the primary VM's virtual disk in a file. This file contains differences with respect to the disk state at the time of the last snapshot operation (during the transition to secure mode). In the Capsule system, this difference file is stored on a partition that resides in main memory, commonly referred to as a RAM disk. Our implementation uses the ImDisk Virtual Disk Driver [Lagerkvist08]. Storing the files on a RAM disk prevents potentially sensitive data that the primary VM writes to its virtual disk from reaching the physical disk. Although malicious access to the physical disk is not the focus of this chapter's threat model, allowing confidential data to touch the disk increases the overall risk of leakage and the cost of hardware destruction.

The main downside to storing virtual disk modifications in memory is that the system requires more memory in secure mode. However, the size of primary disk modifications in secure mode should be small under normal circumstances because they are all temporary. Under normal

المنارة للاستشارات
www.manaraa.com

circumstances, the user has no reason to store big files in a location where they will soon be deleted. If the change buffer does fill up due to a denial-of-service by malware or non-standard usage, then writes to the main virtual disk will fail until the system reverts to normal mode. If there turns out to be a legitimate need for large amounts of temporary storage in secure mode, then the change buffer could be stored on the physical disk and the VMM could use an encrypted file system for added protection.

3.4.5 Reverting to Normal Mode

Transitioning the Capsule system from secure mode back to normal mode is the most security-sensitive operation. Care must be taken to prevent leaks from secure mode back to normal mode insofar as practical for the desired level of security. A full discussion of the channels through which information might flow during this transition and countermeasures can be found in the next section. The Capsule system begins reverting to normal mode when the user enters a key escape sequence. Here, the escape sequence is not to prevent spoofing, but instead to reduce the primary VM's ability to leak data through a timing channel. After the user hits the escape sequence, the following steps take place:

1. The VMM module notifies the Capsule server of the pending transition, which in turn notifies the Capsule viewer.
2. The Capsule server waits up to 30 seconds for the primary VM to flush disk writes to the Storage Capsule. In our experiments, flushing always took less than one second, but uncommon workloads could make it take longer. We chose 30 seconds because it is the default maximum write-back delay for Linux.
3. The secure VM reboots in order to flush any state that was affected by the primary VM. (This blocks some covert channels that are discussed in the next section.)
4. The VMM module halts the primary VM, and then reverts its state to the snapshot taken before entering secure mode and resumes execution.
5. The VMM module re-enables network and other device output for the primary VM.

After the Capsule system has reverted to normal mode, all changes that were made in the primary VM during secure mode, except those to the Storage Capsule, are lost. Also, when the Capsule viewer resumes executing in normal mode, it queries the Capsule to see what mode it is in (if the connection fails due to the reboot, normal mode is assumed). This is a similar mechanism to the return value from a fork operation. Without it, the Capsule viewer cannot tell whether secure mode is just beginning or the system has just reverted to normal mode, because both modes start from the same state.

المنارة للاستشارات

www.manaraa.com

3.4.6 Exporting Storage Capsules

After modifying a storage capsule, the user will probably want to back it up or transfer it to another location at some point. Storage Capsules support this use case by providing an export operation. The Capsule viewer may request an export from the Capsule server at any time during normal mode. When the Capsule server exports an encrypted Storage Capsule back to the primary VM, it is essential that malicious software cannot glean any information from the differences between the Storage Capsule at the time of export and at the original time of import. This should be the case even if malware has full control of the primary VM during secure mode and can manipulate the Storage Capsule contents in a chosen-plaintext attack.

For the Storage Capsule encryption scheme to be secure, the difference between the exported cipher-text and the original imported cipher-text must appear completely random. If the primary VM can change specific parts of the exported Storage Capsule, then it could leak data from secure mode. To combat this attack, the Capsule server re-encrypts the entire Storage Capsule using a new random 128-bit starting tweak value before each export. There is a small chance of two exports colliding. For any two Storage Capsules, each of size 2 GB (2^{27} encryption blocks), the chance of random 128-bit tweak values partially colliding would be approximately 1 in $2 * 2^{27} / 2^{128}$ or 1 in 2^{100}. Because of the birthday paradox, however, there will be a reasonable chance of a collision between a pair of exports after only 2^{50} exports. This number decreases further for Storage Capsules larger than 2 GB. Running 2^{50} exports would still take an extremely long time (36 million years running 1 export / second). We believe that such an attack is unlikely to be an issue in reality, but could be mitigated if future tweaked encryption schemes support 256-bit tweak values.

3.4.7 Key Escape Sequences

During all Capsule operations, the primary VM and the Capsule viewer are not trusted. Some steps in the Capsule system's operation involve the viewer asking the user to enter a key escape sequence. If the primary VM becomes compromised, however, it could just skip asking the user to enter escape sequences and display a spoofed UI that looks like what would show up if the user had hit an escape sequence. This attack would steal the file decryption password while the system is still in normal mode. The defense against this attack is that the user should be accustomed to entering the escape sequence and therefore hit it anyway or notice the anomalous behavior.

It is unclear how susceptible real users would be to a spoofing attack that omits asking for an escape sequence. The success of such an attack is likely to depend on user education. Formally

المنارة للاستشارات
www.manaraa.com

evaluating the usability of escape sequences in the Capsule system is future work. Another design alternative that may help if spoofing attacks are found to be a problem is reserving a secure area on the display. This area would always tell the user whether the system is in secure mode or if the secure VM has control of the UI.

3.5 Covert Channel Analysis

The Storage Capsule system is designed to prevent any direct flow of information from secure mode to normal mode. However, there are a number of covert channels through which information may be able to persist during the transition from secure to normal mode. This section tries to answer the following questions about covert channels in the Capsule system as best as possible:

- Where can the primary virtual machine store information that it can retrieve after reverting to normal mode?
- What defenses might fully or partially mitigate these covert information channels?

We do not claim to expose all covert channels here, but list many channels that have been uncovered during this research. Likewise, the proposed mitigation strategies are not necessarily optimal, but represent possible approaches for reducing the bandwidth of covert channels. Measuring the maximum bandwidth of each covert channel requires extensive analysis and is beyond the scope of this work. There has been a great deal of research on measuring the bandwidth of covert channels [Browne94, Kang95, Moskowitz94, Percival05, Trostle91, Wang06], which could be applied to calculate the severity of covert channels in the Capsule system in future work.

The covert channels discussed in this section can be divided into four categories:

1. Primary OS and Capsule – Specific to Storage Capsule design
2. External Devices – Includes floppy, CD-ROM, USB, SCSI, etc.
3. VMM – Arising from virtual machine monitor implementation, includes memory mapping and virtual devices
4. Core Hardware – Includes CPU and disk drives

The Capsule system prevents most covert channels in the first two categories. It can use the VMM to mediate the primary virtual machine's device access and completely erase the primary VM's state when reverting to normal mode. The Capsule system also works to mitigate timing channels when switching between modes of operation.

www.manaraa.com

Storage Capsules do not necessarily protect against covert channels in the last two categories. There has been a lot of work on identifying, measuring, and mitigating covert channels in core hardware for traditional MLS systems [Kang95, Kemmerer83, Moskowitz94, Trostle91]. Similar methods for measuring and mitigating those core channels could be applied to Storage Capsules. Covert channels arising from virtualization technology have not received much attention. This research hopes to highlight some of the key mechanisms in a VMM that can facilitate covert communication. The remainder of this section explores covert channels in each of these categories, including mitigation strategies and their required trade-offs.

This discussion of covert channels also applies to most side channels, which have a weaker threat model. In a side channel attack, the hacker would obtain some residual secret information from secure mode, *without* collusion of the primary VM during secure mode. An example of a side channel attack would be recovering bits of the file encryption key by monitoring re-encryption time during an export operation. Previous work looks at side channels in cryptographic algorithms [Kocher96]. Side channels may exist in the Storage Capsule system, but we do not explicitly evaluate them here. Exploiting side channels is likely to be more difficult than exploiting covert channels, which only requires compromising the unprotected and low-integrity Primary VM.

3.5.1 Primary OS and Storage Capsule Channels

In addition to covert channels that one might find in other systems, the design of Storage Capsules creates opportunities for covert communication. This section focuses on the interactions between the primary VM and the secure VM. The primary VM should not be able to affect the secure VM's state during secure mode in a way that it can detect during normal mode. Here we examine three attack vectors that a compromised primary VM might use to steal data, and corresponding counter-measures.

Transition Timing

When the VMM restores the primary virtual machine from a snapshot, the restored state contains the exact same information as prior to entering secure mode. Following the restoration, the primary VM can obtain additional information by fetching the system time, which tells it when the system returned to normal mode. If the primary VM can manipulate the transition timing, then it can leak data.

The Capsule system prevents direct leaks through transition timing by requiring a secure key escape sequence to switch back to normal mode. However, the primary VM may still be able

المنارة للاستشارات
www.manaraa.com

to affect transition timing depending on how long the Capsule server waits for the primary VM to flush disk writes to the Storage Capsule. One option that would preclude manipulation of transition timing would be for the Capsule server to wait a fixed time-period for disk writes to complete. This would be the most secure approach. Unfortunately, this strategy would either impact usability or threaten correctness. If the server were to stall 30 seconds – the default write-back delay for Linux – then the transition time would be unnecessarily long in most cases. On the other hand, if it only waited a few seconds, then some delayed writes might fail and corrupt the Storage Capsule.

For our implementation of the Capsule system, we chose a transition timing policy that sacrifices a few bits of potential information leakage per transition, but is more usable. The Capsule server will wait for the primary VM to flush Storage Capsule writes for up to 30 seconds. If the primary VM finishes the disk flush in less than 30 seconds, the server will immediately proceed with restoration. This opens up the possibility of the primary VM stalling after flushing writes so that it can read back timing information after returning to normal mode. The maximum bandwidth of this timing channel is somewhat limited, however, because the snapshot restoration adds noise to the transition time. Depending on how much memory the primary VM has, the amount of time between exiting secure mode and resuming in normal mode can vary by several seconds. By notifying the server at a specific offset from the most recent half-minute, the primary VM will only be able to communicate a few bits of data. One possible counter-measure to this type of timing channel would be to deliberately skew the clock during secure mode so the primary VM cannot accurately judge system time.

Storage Capsule Contents

The ideal place for a compromised primary VM to store data would be in the Storage Capsule itself. It is essential that the Capsule server cut off this channel. First, Storage Capsules must have a fixed size. This imposes a small usability limitation, which we believe is reasonable given that other popular systems like TrueCrypt [TrueCrypt09] fix the size of encrypted file containers. Enforcing the next constraint required to cut off storage channels is slightly more complicated. No matter what changes the primary VM makes to the Storage Capsule in secure mode, it must not be able to deduce what has been changed after the Capsule server exports the Storage Capsule. As discussed earlier, XTS-AES encryption with a different tweak value for each export satisfies this requirement. Whether the primary VM changes every single byte or does not touch anything, the resulting exported Storage Capsule will be random with respect to its original contents.

المنارة للاستشارات
www.manaraa.com

Social Engineering Attacks

If the primary virtual machine cannot find a way to leak data directly, then it can resort to influencing user behavior. The most straightforward example of a social engineering attack would be for the primary VM to deny service to the user by crashing at a specific time, and then measuring transition time back to normal mode. There is a pretty good chance that the user would respond to a crash by switching back to normal mode immediately, especially if the system is prone to crashing under normal circumstances. In this case, the user may not even realize that an attack is taking place. Another attack that is higher-bandwidth, but perhaps more suspicious, would be for the primary VM to display a message in secure mode that asks the user to perform a task that leaks information. For example, a message could read “Automatic update failed, please open the update dialog and enter last scan time ‘4:52 PM’ when internet connectivity is restored.” Users who do not understand covert channels could easily fall victim to this attack. In general, social engineering is difficult to prevent. The Capsule system currently does not include any counter-measures to social engineering. In a real deployment, the best method of fighting covert channels would be to properly educate the users.

3.5.2 External Device Channels

Any device that is connected to a computer could potentially store information. Fortunately, most devices in a virtual machine are virtual devices, including the keyboard, mouse, network card, display, and disk. In a traditional system, two processes that have access to the keyboard could leak data through the caps-, num-, and scroll-lock state. The VMware VMM resets this device state when reverting to a snapshot, so a virtual machine cannot use it for leaking data. We did not test virtualization software other than VMware to see how it resets virtual device state.

Some optional devices may be available to virtual machines. These include floppy drives, CD-ROM drives, sound adapters, parallel ports, serial ports, SCSI devices, and USB devices. In general, there is no way of stopping a VM that is allowed to access these devices from leaking data. Even devices that appear to be read-only, such as a CD-ROM drive, may be able to store information. A VM could eject the drive or position the laser lens in a particular spot right before switching back to normal mode. While these channels would be easy to mitigate by adding noise, the problem worsens when considering a generic bus like USB. A USB device could store anything or be anything, including a disk drive. One could allow access to truly read-only devices, but each device would have to be examined on an individual basis to check for covert channels. The Capsule system prevents these covert channels because the primary VM is not

المنارة للاستشارات
www.manaraa.com

given access to external devices. If the primary VM needs access to external devices, then they would have to be disabled during secure mode.

3.5.3 Virtual Machine Monitor Channels

In a virtualization system, everything is governed by the virtual machine monitor, including memory mapping, device I/O, networking, and snapshot saving/restoration. The VMM's behavior can potentially open up new covert channels that are not present in a standard operating system. These covert channels are implementation-dependent and may or may not be present in different VMMs. This section serves as a starting point for thinking about covert channels in virtual machine monitors.

Memory Paging

Virtual machines are presented with a virtual view of their physical memory. From a VM's perspective, it has access to a contiguous "physical" memory segment with a fixed size. When a VM references its memory, the VMM takes care of mapping that reference to a real physical page, which is commonly called a machine page. There are a few different ways that a VMM can implement this mapping. First, it could directly pin all of the virtual machine's physical pages to machine pages. If the VMM uses this strategy, and it keeps the page mapping constant during secure mode and after restoration, then there is no way for a virtual machine to affect physical memory layout. However, this fixed mapping strategy is not always the most efficient way to manage memory.

Prior research describes resource management strategies in which the VMM may over-commit memory to virtual machines and page some of the VM's "physical" memory out to disk [Govil99, Waldspurger02]. If the VMM employs this strategy, then a virtual machine can affect the VMM's page table by touching different pages within its address space. The residual effects of page table manipulation may be visible to a VM after a snapshot restoration, unless the VMM first pages in all of the VM's memory. A snapshot restoration should page in all of a VM's memory in most cases. But, if it is a "background" restoration, then accessing a memory location that has not been loaded from the snapshot yet *and* is paged out to disk might incur two page faults instead of one, which can be measured and may leak information.

VMware ESX server employs a number of tricks that allow it to over-commit memory to virtual machines [Waldspurger02]. Each of the mechanisms described by Waldspurger allow efficient server consolidation, but also create an opportunity for covert channels. The first mechanism he describes is *ballooning*. Guest virtual machines contain balloon drivers that

المنارة للاستشارات
www.manaraa.com

allocate extra memory and then tell the VMM it can page out the corresponding “physical” pages. A misbehaving guest OS can exploit ballooning by touching the balloon driver’s pages and causing the VMM to load them in from disk. When the VM is subsequently reverted to a snapshot, the compromised OS can again touch all of the balloon driver pages to see whether they have already been faulted in prior to the snapshot restoration. The VMM could mitigate this ballooning channel by restoring the state of its page table to match the original set of ballooned pages when restoring a snapshot. Of course, ballooning may be used as a covert channel between simultaneously executing VMs in systems other than Capsule.

There has also been research on sharing memory pages with the same content in virtual machines [Bugnion97, Waldspurger02]. If the VMM maps multiple pages to one machine page and marks them as copy-on-write, then the virtual machine will encounter a page fault when writing to one of them. If a VM sets a large number of pages to have the same value during secure mode, then a much larger number of page faults will take place when restoring a snapshot. However, these page faults will only be noticeable to the VM if execution resumes in normal mode before all of the VM’s memory pages are loaded from the snapshot file. In this case, the VM can measure the total restoration time or write to pages before they have loaded and test for page faults to recover information. If the VM resumes execution after its memory has been fully restored and pages have been re-scanned for duplication, then this covert channel will not work.

The Capsule system does not over-commit memory for virtual machines, so the memory saving techniques mentioned above do not take effect. Our implementation of the Capsule system does not employ any counter-measures to covert channels based on memory paging.

Virtual Networks

The Capsule system blocks external network access during secure mode, but it relies on a virtual network for communication between the secure VM and the primary VM. While the virtual network itself is stateless (to the best of our knowledge), anything connected to the network could potentially be a target for relaying information from secure mode to normal mode. The DHCP and NAT services in the VMM are of particular interest. A compromised virtual machine may send arbitrary packets to these services in an attempt to affect their state. For example, a VM might be able to claim several IP addresses with different spoofed MAC addresses. It could then send ARP requests to the DHCP service following snapshot restoration to retrieve the spoofed MAC addresses, which contain arbitrary data. The Capsule system restarts both the DHCP and NAT services when switching back to normal mode to avoid this covert channel.

المنارة للاستشارات
www.manaraa.com

Any system that allows both a high-security and low-security VM to talk to a third trusted VM (e.g., secure VM in Capsule) exposes itself another covert channel. Naturally, all bets are off if the primary VM can compromise the secure VM. Even assuming the secure VM is not vulnerable to remote exploitation, it still may be manipulated to relay data from secure mode back to normal mode. Like the DHCP service on the host, the secure VM's network stack stores information. For example, the primary VM could send out TCP SYN packets with specific source port numbers that contain several bits of data right before reverting to normal mode. When the primary VM resumes execution, it could see the source ports in SYN/ACK packets from the secure VM.

It is unclear exactly how much data can be stashed in the network stack on an unsuspecting machine and how long that information will persist. The only way to guarantee that a machine will not inadvertently relay state over the network is to reboot it. This is the approach we take to flush the secure VM's network stack state when switching back to normal mode in Capsule.

Guest Additions

The VMware VMM supports additional software that can run inside of virtual machines to enhance the virtualization experience. The features of guest additions include drag-and-drop between VMs and a shared clipboard. These additional features would undermine the security of any virtual machine system with multiple confidentiality levels and are disabled in the Capsule system.

3.5.4 Core Hardware Channels

Core hardware channels allow covert communication via one of the required primary devices: CPU or disk. Memory is a core device, but memory mapping is handled by the VMM, and is discussed in the previous section. Core hardware channels might exist in any multi-level secure system and are not specific to Storage Capsules or virtual machines. One difference between prior research and this work is that prior research focuses on a threat model of two processes that are executing simultaneously on the same hardware. In the Capsule system, the concern is not with simultaneous processes, but with a low-security process (normal-mode VM) executing on the same hardware after a high-security process (secure-mode VM) has terminated. This constraint rules out some traditional covert channels that rely on resource contention, such as a CPU utilization channel.

المنارة للاستشارات
www.manaraa.com

CPU State

Restoring a virtual machine's state from a snapshot will overwrite all of the CPU register values. However, modern processors are complex and store information in a variety of persistent locations other than architecture registers. Many of these storage areas, such as branch prediction tables, are not well-documented or exposed directly to the operating system. The primary method for extracting this state is to execute instructions that take a variable number of clock cycles depending on the state and measure their execution time, or exploit speculative execution feedback. Prior research describes how one can use these methods to leak information through cache misses [Percival05, Wang06].

There are a number of counter-measures to covert communication through CPU state on modern processors. In general, the more instructions that execute in between secure mode and normal mode, the less state will persist. Because the internal state of a microprocessor is not completely documented, it is unclear exactly how much code would need to run to eliminate all CPU state. One guaranteed method of wiping out all CPU state is to power off the processor. However, recent research on cold boot attacks [Halderman08] shows that it may take several minutes for memory to fully discharge. This strategy would lead to an unreasonably long delay when switching from secure mode to normal mode.

The ideal solution for eliminating covert CPU state channels in Capsule and other virtualization systems would be with hardware support. The latest CPUs already support hardware virtualization, which allows them to fully emulate instruction sets for virtual machines. An additional mechanism is needed when switching between virtual machines that not only restores register and memory mappings, but also restores all state that could affect VM execution. This operation would load all of the cache data (L1, L2, and instruction), the branch prediction table, and any other intermediate state. It would also be sufficient to provide an instruction that would erase all of this data.

Although the Capsule system does not take counter-measures to prevent CPU state covert channels, the VMM restores a significant portion of the primary VM's original memory state before the VM can resume execution. This restoration will likely eliminate any residual cache lines from secure mode. It will also add a great deal of noise to any other CPU state, such as branch prediction tables, due to the number of instructions that execute during restoration. Computing the maximum bandwidth of CPU state channels in the Capsule system is future work.

المنارة للاستشارات
www.manaraa.com

Disk State

The caching and layout of files on disk, and in the file system, can be used to transmit information. Another covert channel exists in the positioning of the disk arm [Gold79]. If the primary VM can manipulate disk state in a way that is readable after transitioning back to normal mode, then it can leak data. There are a few aspects of the Capsule system's design that make such an attack much more difficult. Any writes to the virtual disk during secure mode actually go to flat difference file that stores changes since the last snapshot operation. The primary VM has no control over where the data goes on disk, only how much is written., which would make exploitation of such a channel difficult. The VMM also deletes this file when transitioning back to normal mode. This frees up any blocks that were allocated for the difference file. Finally, the VMM reads in the primary VM's original memory state from disk during snapshot restoration. This moves the position of the disk arm and adds considerable noise to the cache.

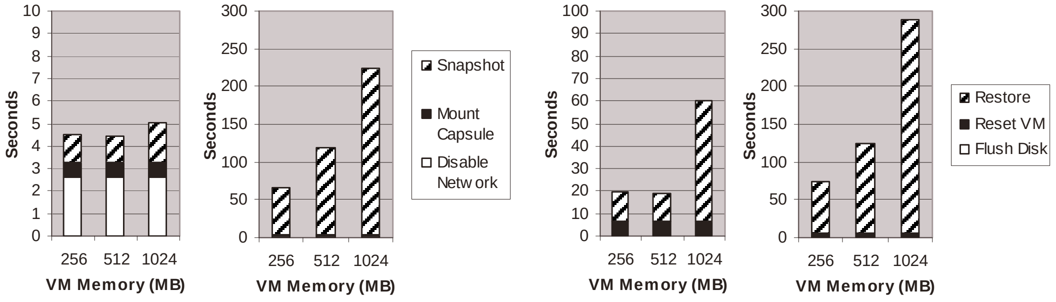
Despite the noise from transitioning back to normal mode, the disk cache is a likely candidate for a covert channel. All writes are sent to a difference file and deleted, but reads touch the same virtual disk file during secure mode and normal mode. Consider the primary VM sequentially reading every block on its disk during secure mode to transmit a 1, or reading very little to transmit a 0. The primary VM might be able to tell what happened in secure mode by reading back disk blocks and measuring if they hit the disk cache. One would need to flush all of the disk caches to eliminate this channel. The Capsule system does not take any steps to mitigate disk state channels.

3.5.5 Mitigating VMM and Core Hardware Covert Channels

The design of Storage Capsules centers around improving local file encryption with a minimal impact on existing behavior. The user has to take only a few additional steps, and no new hardware is required. The current implementation is designed to guard against many covert channels, but does not stop all of them, such as the CPU state, through which data may leak from secure to normal mode. If the cost of small leaks outweighs usability and the cost of extra hardware, then there is an alternative design that can provide additional security.

One way of cutting off almost all covert channels would be to migrate the primary VM to a new isolated computer upon entering secure mode. This way, the virtual machine would be running on different core hardware and a different VMM while in secure mode, thus cutting off covert channels at those layers. VMware ESX server already supports live migration, whereby a virtual machine can switch from one physical computer to another without stopping execution. The user would have two computers at his or her desk, and use one for running the primary VM

www.manaraa.com



(a) **(b)** **(c)** **(d)**

Figure 3.2. Transition times for different amounts of primary VM memory.
(a) to secure mode with background snapshot, (b) to secure mode with full snapshot,
(c) to normal mode with background restore, and (d) to normal mode with full restore.

in secure mode, and the other for normal mode. When the user is done accessing a Storage Capsule, the secure mode computer would reboot and then make the Storage Capsule available for export over the network. This extension of the Capsule system's design would drastically reduce the overall threat of covert channels, but would requires additional hardware and could add usability impediments that would not be suitable in many environments.

3.6 Performance Evaluation

There are three aspects of performance that are important for Storage Capsules: (1) transition time to secure mode, (2) system performance in secure mode, and (3) transition time to normal mode. It is important for transitions to impose only minimal wait time on the user and for performance during secure mode to be comparable to that of a standard computer for common tasks. This section evaluates Storage Capsule performance for transitions and during secure mode. The experiments were conducted on a personal laptop with a 2 Ghz Intel T2500 processor, 2 GB of RAM, and a 5200 RPM hard drive. Both the host and guest operating systems (for the secure VM and primary VM) were Windows XP Service Pack 3, and the VMM software was VMware Workstation ACE Edition 6.0.4. The secure VM and the primary VM were both configured with 512 MB of RAM and to utilize two processors, except where indicated otherwise.

The actual size of the Storage Capsule does not affect any of the performance numbers in this section. It does, however, influence how long it takes to run an import or export. Both import and export operations are expected to be relatively rare in most cases – import only occurs when loading a Storage Capsule from an external location, and export is required only when transfering a Storage Capsule to another user or machine. Importing and exporting consist of a disk read,

www.manaraa.com

encryption (for export only), a local network transfer, and a disk write. On our test system, the primary VM could import a 256 MB Storage Capsule in approximately 45 seconds and export it in approximately 65 seconds. Storage Capsules that are imported and exported more often, such as e-mail attachments, are likely to be much smaller and should take only a few seconds.

3.6.1 Transitioning to and from Secure Mode

The transitions to and from secure mode consist of several tasks. These include disabling/enabling device output, mounting/dismounting the Storage Capsule, saving/restoring snapshots, waiting for an escape sequence, and obtaining the encryption key. Fortunately, some operations can happen in parallel. During the transition to secure mode, the system can do other things while waiting for user input. The evaluation does not count this time, but it will reduce the delay experienced by the user in a real deployment. VMware also supports both background snapshots (copy-on-write) and background restores (copy-on-read). This means that execution may resume in the primary VM before memory has been fully saved or restored from the snapshot file. The system will run slightly slower at first due to page faults, but will speed up as the snapshot or restore operation nears completion. A background snapshot or restore must complete before another snapshot or restore operation can begin. This means that even if the primary VM is immediately usable in secure mode, the system cannot revert to normal mode until the snapshot is finished.

Figure 3.2 shows the amount of time required for transitioning to and from secure mode with different amounts of RAM in the primary VM. Background snapshots and restorations make a huge difference. Transitioning to secure mode takes 4 to 5 seconds with a background snapshot, and 60 to 230 seconds without. The time required for background snapshots, mounting the Storage Capsule, and disabling network output also stays fairly constant with respect to primary VM memory size. However, the full snapshot time scales linearly with the amount of memory. Note that the user must wait for the full snapshot time before reverting to normal mode.

The experiments show that reverting to normal mode is a more costly operation than switching to secure mode, especially when comparing the background restore to the background snapshot operation. This is because VMware allows a virtual machine to resume immediately during a background snapshot, but waits until a certain percentage of memory has been loaded in a background restore. Presumably, memory reads are more common than memory writes, so copy-on-read for the restore has worse performance than copy-on-write for the snapshot. VMware also appears to employ a non-linear strategy for deciding what portion of a background restore must complete before the VM may resume execution. It waited approximately the same

المنارة للاستشارات
www.manaraa.com

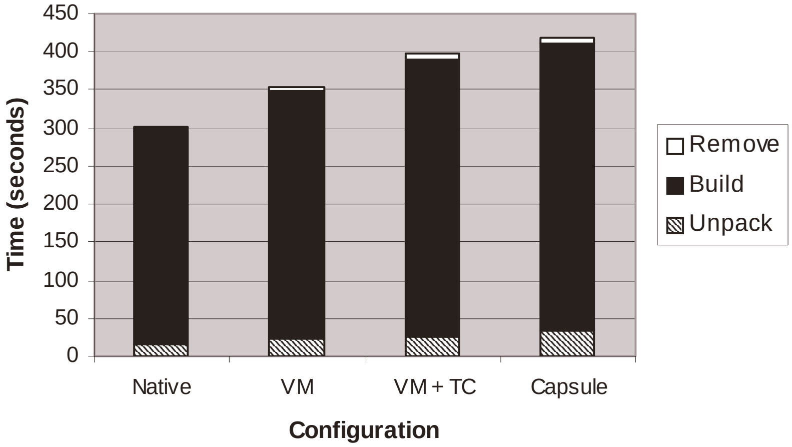


Figure 3.1. Results from building Apache with a native OS, a virtual machine, a virtual machine running TrueCrypt, and Capsule. Storage Capsules add only a 5% overhead compared to a VM with TrueCrypt, 18% compared to a plain VM, and 38% overhead compared to a native OS.

amount of time when a VM had 256 MB or 512 MB of RAM, but delayed significantly longer for the 1 GB case.

The total transition times to secure mode are all reasonable. Many applications will take 4 or 5 seconds to load a document anyway, so this wait time imposes little burden on the user. The transition times back to normal mode for 256 MB and 512 MB are also reasonable. Waiting less than 20 seconds does not significantly disrupt the flow of work. However, 60 seconds may be long wait time for some users. It may be possible to optimize snapshot restoration by using copy-on-write memory while the primary VM is in secure mode. This way, the original memory would stay in tact and the VMM would only need to discard changes when transitioning to normal mode. Optimizing transition times in this manner is future work.

3.6.2 Performance in Secure Mode

Accessing a Storage Capsule imposes some overhead compared to a normal disk. A Storage Capsule read or write request traverses the file system in the primary VM, and is then sent to the secure VM over the virtual network. The request then travels through a layer of encryption on the secure VM, out to its virtual disk, and then to the physical drive. We compared the disk and processing performance of Storage Capsules to three other configurations. These configurations consisted of a native operating system, a virtual machine, and a virtual machine with a TrueCrypt encrypted file container. For the evaluation, we ran an Apache build benchmark. This benchmark involves decompressing and extracting the Apache web server source code, building the code, and then removing all of the files. The Apache build benchmark

المنارة للاستشارات
www.manaraa.com

probably represents the worst case scenario for Storage Capsule usage. We expect that the primary use of Storage Capsules will be for less disk-intensive activities like editing documents or images, for which the overhead should be unnoticeable.

Figure 3 shows the results of the Apache build benchmark. Storage Capsules performed well overall, only running 38% slower than a native system. Compared to a single virtual machine running similar encryption software (TrueCrypt), Storage Capsules add an overhead of only 5.1% in the overall benchmark and 31% in the unpack phase. This shows that transferring reads and writes over the virtual network to another VM has a reasonably small performance penalty. The most significant difference can be seen in the remove phase of the benchmark. It executes in 1.9 seconds on a native system, while taking 5.5 seconds on a VM, 6.5 seconds on a VM with TrueCrypt, and 7.1 seconds with Storage Capsules. The results from the VM and VM with TrueCrypt tests show, however, that the slowdown during the remove phase is due primarily to disk performance limitations in virtual machines rather than the Capsule system itself.

3.7 Conclusion and Future Work

This chapter introduced Storage Capsules, a new mechanism for securing files on a personal computer. Storage Capsules are similar to existing encrypted file containers, but protect sensitive data from malicious software during decryption and editing. The Capsule system provides this protection by isolating the user's primary operating system in a virtual machine. The Capsule system turns off the primary OS's device output while it is accessing confidential files, and reverts its state to a snapshot taken prior to editing when it is finished. One major benefit of Storage Capsules is that they work with current applications running on commodity operating systems.

Covert channels are a serious concern for Storage Capsules. This research explores covert channels at the hardware layer, at the VMM layer, in external devices, and in the Capsule system itself. It looks at both new and previously examined covert channels from a novel perspective, because Storage Capsules have different properties than side-by-side processes in a traditional multi-level secure system. The research also suggests ways of mitigating covert channels and highlights their usability and performance trade-offs. Finally, we evaluated the overhead of Storage Capsules compared to both a native system and standard virtual machines. We found that transitions to and from secure mode were reasonably fast, taking 5 seconds and 20 seconds, respectively. Storage Capsules also performed well in an Apache build benchmark, adding 38%

المنارة للاستشارات
www.manaraa.com

overhead compared to a native OS, but only a 5% penalty when compared to running current encryption software inside of a virtual machine.

In the future, we plan to further explore covert channels discussed in this work. This includes measuring their severity and quantifying the effectiveness of mitigation strategies. We also hope to conduct a study on usability of keyboard escape sequences for security applications. Storage Capsules rely on escape sequences to prevent spoofing attacks by malicious software, and it would be beneficial to know how many users of the Capsule system would still be vulnerable to such attacks.

للاستشارات المنارة
www.manaraa.com

CHAPTER 4

NETWORK-BASED CONFIDENTIALITY THREAT DETECTION

4.1 Overview

Proactive controls, both at the host and the network level, are not enough to protect against all security threats. As a society and a research community, we are far form adequately securing computers that house confidential information. Host-level systems with strict security controls, such as Capsule, which was presented in the previous chapter, are only able to provide security guarantees in a limited set of scenarios. Furthermore, host-level security software deployment incurs a significant management overhead and simply may not happen due to cost or lack of expertise.

Current network-based security systems, such as firewalls and intrusion detection systems (IDSs), do an incomplete job of mitigating threats that slip past host-level controls. Many of them operate on network traffic at the transport layer and are unable to identify malicious traffic that falls within the domain of acceptable activity from a network perspective, such as downloading malicious software sent in an e-mail message. Those that do inspect application layer information are fundamentally limited by their approach of directly searching for malicious activity. In order to effectively search for bad network traffic, one must know what it looks like ahead of time, which is difficult for the latest threats, which are rapidly evolving.

Here, we begin to address the problem of detecting malicious activity over the network by classifying network traffic based on its source application. As the size and diversity of the Internet grows, so do the applications that use the network. Originally, network applications such as web browsers, terminal clients, and e-mail readers were the only programs accessing the Internet. Now, almost every application has a networking component, whether it is to obtain updates, manage licensing, or report usage statistics. Although pervasive network connectivity provides a number of benefits, it also introduces security risks. In addition to software that is

المنارة للاستشارات
www.manaraa.com

outright malicious, many programs that access the network allow users to leak confidential information or expose them to new attack vectors. An example is instant messaging (IM) software. Most IM programs permit direct file transfers. Also, so-called IM viruses are able to circumvent security systems by going through the IM network itself [Mannan05]. Peer-to-peer file sharing software presents a risk as well because files often come packaged with Trojan horse malware. Being able to categorize network traffic and determine its origin, including traffic from *unwanted* programs that are not directly malicious, is a critical first step in effectively searching for confidentiality threats.

In this chapter, we present methods for detecting network applications by only looking at their web traffic. These methods differentiate programmatic web service access from human web browsing and expose information about active network applications. The methods focus on two key aspects of web traffic: *timing* and *formatting*. Human web requests occur in randomly interspersed bursts. Programmatic web requests, on the other hand, will often happen at regular fixed intervals. Humans also tend to browse the web at specific times according to a schedule, while programs may access the web at any hour of the day or night. We take advantage of this knowledge to discover network applications that call home on a regular basis.

Another web traffic characteristic that we examine in this chapter is formatting. The HTTP protocol specification contains a “User-Agent” field that applications may use to identify themselves. Although most malicious programs do not identify themselves as such, they will often select a user-agent value that does not match any legitimate programs or omit the field entirely. HTTP also allows for application-specific extension header fields that may hold arbitrary information. The presence of a new or different header field in a request indicates that it came from a non-browser network application. Again, some malware tries to mimic a web browser in its traffic formatting. However, malware writers are prone to human error and may accidentally include an anomalous header field. An example is a spyware program that mistakenly spelled an HTTP header field “referrer” (the correct dictionary spelling), while the specification states that it should be spelled “referer,” an incorrect spelling. Our analysis allows us to quickly detect network applications with unique message formatting, which includes a significant amount of malicious software.

We evaluated the accuracy of our timing and formatting analyses by testing them on web traffic collected from 30 users over a 40-day period. The algorithms presented in this chapter were able to effectively detect programmatic web access while generating few false positives. Of the total alerts, only 7% were false positives, an average of about one per day. We believe this to be an acceptable false positive rate given the number of computers in the study.

www.manaraa.com

In this chapter, we chose to focus on HTTP traffic. The reasons for this are twofold. First, HTTP is the most widely-allowed network protocol, and is often the only way to communicate data out to the Internet through firewalls and proxy servers. Second, the HTTP protocol probably has more implementations by more different applications than any other protocol. Protocols such as secure shell and FTP are usually only implemented by secure shell and FTP utilities, not by other programs that only use the Internet for registration, licensing, and updates.

The rest of this chapter is laid out as follows. Section 4.2 discusses related work on detecting network applications. Section 4.3 describes the timing analysis methods. Section 4.4 presents the formatting analysis techniques. Section 4.5 shows the results from our test deployment. Section 4.6 includes results from an evaluation against known tunneling programs. Section 4.7 talks about how a hacker could bypass the analysis techniques presented in this chapter. Finally section 4.8 concludes and discusses future work.

4.2 Related Work

Network monitoring systems exist that can differentiate between traffic of different protocols regardless of the transport-layer port [Netwitness09, Sandvine09]. These systems can even identify protocol tunneling. (Tunneling is when one application-layer protocol is embedded within the payload of another, e.g., HTTPS is HTTP tunneled over SSL.) In this chapter, we go a step further by identifying different applications that implement the *same* protocol. This is a more difficult problem because the differences between implementations of the same protocol are much more subtle.

Brumley et al. present a system for automatically discovering deviations between implementations of the same network protocol [Brumley07]. Their approach involves binary analysis of the programs in question. In contrast, the algorithms presented in this chapter can differentiate between different applications that use the same protocol (HTTP) without any prior knowledge of the applications or access to the computers on which they are executing. Furthermore, the system presented by Brumley et al. can only generate network inputs that expose implementation differences, typically in network servers. While the network applications we consider in this chapter may take network inputs, our algorithms instead passively examine the network outputs to differentiate client network applications from one another.

Zhang and Paxson describe a method for detecting backdoors [Zhang00]. They look at the timing of packet arrivals and packet sizes in order to characterize an interactive shell. Others have observed that keystroke inter-arrival periods follow a Pareto distribution [Danzig92]. The

المنارة للاستشارات
www.manaraa.com

Pareto model does not extend to spyware and unwanted network applications that we hope to identify in this chapter because they communicate automatically. The Pareto model only works for traffic that corresponds to human key-presses. Instead, delay times from network applications will follow a distribution according to the callback algorithm chosen by the programmer. Focusing on connections with small packet sizes does not help identify network applications either; they can and do send messages with arbitrary sizes.

Significant research exists on characterizing human browsing patterns to enhance proxy cache and web server performance [Barford98, Duska97, Kelly02]. We look at some of the same traffic characteristics in this chapter. However, the purpose of our analysis is identifying non-human web traffic, not improving server performance.

Kruegel et al. outline a method for detecting malicious web requests at the web server [Kruegel03]. They build a probabilistic profile of web request parameters and detect deviations from this profile. The methods presented in this chapter are different because they aim to identify different network applications by looking at client-side traffic. Due to the diversity of websites on the Internet, it would not be feasible to build a profile for normal web browsing to servers in this manner.

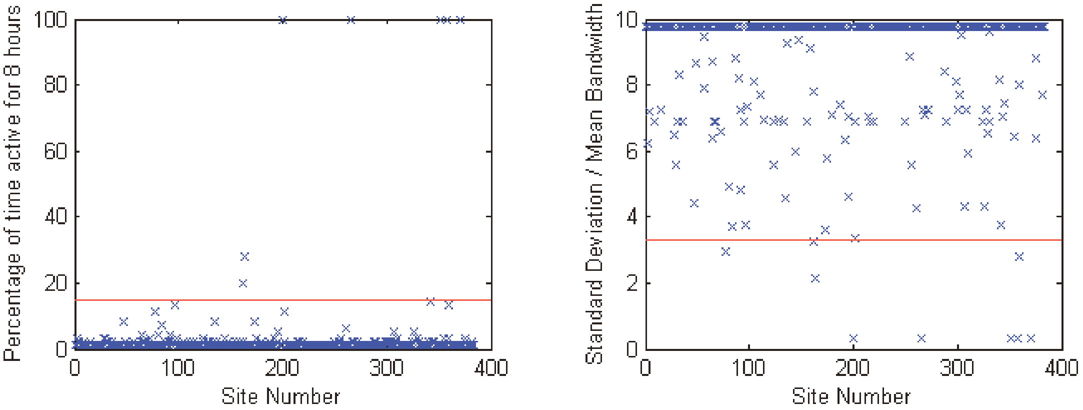
4.3 Timing Analysis

In this section, we explore two methods for differentiating between network messages resulting from human input, and those generated by automated processes. We created these algorithms based on observations from real web traffic for 30 users over a one-week period. Both of these methods consider groups of HTTP messages between each client and each server as separate. The messages between a particular client and server need not all have come from an automated process; the following algorithms are designed to identify HTTP traffic from which *any* of the requests are automatic. Furthermore, if an application communicates with multiple web servers, then these techniques will identify each server that receives automated requests.

4.3.1 Regularity

Human web browsing tends to occur in short bursts. Automated web requests, on the other hand, happen at more regular intervals. We measure request regularity by looking at the amount of outbound bandwidth during 5-minute intervals over the past 8 hours and 48 hours. The goal of measuring regularity is to expose automated web requests, even if they occur at random intervals and are interlaced with human activity.

المنارة للاستشارات
www.manaraa.com



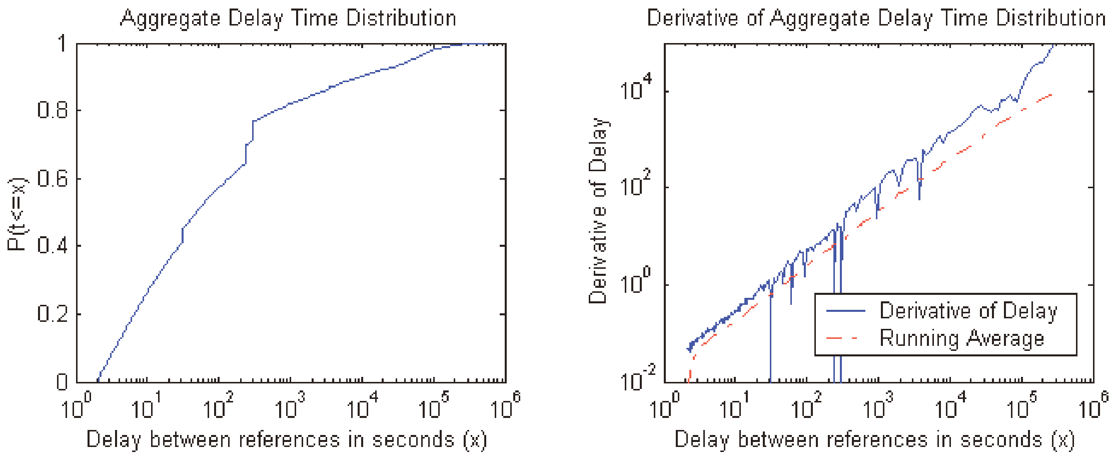
(a) **(b)**

Figure 4.1. (a) Seven sites were detected by usage counts for ~400 sites over 8 hours with a detection threshold of 16% (b) Nine Sites were detected using the deviation over mean during an 8 hour period with a detection threshold of 3.3

Here, we present two methods for computing regularity. The first is to count the number of five-minute time periods during which at least one request was seen for a particular site. If requests appear too often, then they are probably coming from an automated process. Figure 4.1a shows a plot of bandwidth counts over an 8-hour time period for approximately 400 sites accessed by a single user. Using a threshold of 16% activity (80+ minutes), we were able to identify seven sites receiving automated web requests with no false positives. Five of these were in blatant violation of the threshold; they were active during every 5-minute interval in the 8-hour time period. The two other websites served periodically-refreshing advertisements. There were also five servers close to the detection threshold (between 10% and 16%), only two of which were false positives, both from social networking sites. The 16% threshold was chosen conservatively to avoid false alarms and could be lowered even further depending on the target network.

The second method for computing regularity involves calculating the coefficient of variation (c.o.v.) for 5-minute bandwidth measurements over the previous 8 or 48 hours. The coefficient of variation is the standard deviation divided by the mean bandwidth usage. Conceptually, this number represents a normalized deviation. If requests occur in short bursts, which is characteristic of normal human activity, then the coefficient of variation will be high. Low variation in bandwidth usage is indicative of automated activity. The plot of the coefficient of variation measurements for an 8-hour time period can been seen in Figure 4.1b. We found thresholds of 3.3 for 8 hours and 4.5 for 48 hours to be effective. At those settings, the coefficient of variation method detected nine sites in violation of the threshold both over an 8-hour and a 48-hour period, none of which were false positives. Much like the sites close to the threshold for the

المنارة للاستشارات
www.manaraa.com



(a) **(b)**

Figure 4.2. (a) Aggregate delay-time CDF with jumps at t = 30 seconds, 4 minutes, and 5 minutes. (b) Y-derivative of CDF and running average used to detect anomalies.

counting method, three of the five sites just above 3.3 were false positives, all associated with social networking sites. For a network with less frequent browsing, the threshold for this filter could be effectively raised to around 4.0 for an 8-hour period without producing many false positives. All of the seven sites filtered by the counting method were also filtered by the coefficient of variation method.

The reason for employing two different algorithms to measure regularity is that they each have advantages in different circumstances. The c.o.v. measurement is generally more effective for differentiating between human and automated requests. However, a malicious network application could remain active during every 5-minute time interval over an 8-hour period without exceeding the c.o.v. threshold by varying the amount of bandwidth in each interval. This type of attack would fail if both the count and c.o.v. algorithms are deployed.

4.3.2 Inter-Request Delay

Some network applications generate requests using a fixed-interval timer. The goal of our delay-time measurements is to identify timer-driven requests. We measured the inter-request delay time for requests to each server from each client. We stored these delay measurements in individual vectors for each client/server pair and in an aggregate vector for all clients and servers to observe the general inter-request delay distribution. Figure 4.2a shows the probabilistic distribution of all delay times. You can notice jumps in the cumulative distribution function (CDF) at 30 seconds, 4 minutes, and 5 minutes. There are also less-pronounced jumps at 15

www.manaraa.com

$$\text{Derivative of Delay} = \begin{cases} 0 & t < 2 \\ V(t) - V(t-1) & 2 \le t \le Size(V) \end{cases}$$

$$\text{Running Average} = \frac{.8 * \sum_{i=1}^{a} Derivative(t-i)}{a} \qquad 1 \le t \le Size(V)$$

Figure 4.3. Equations for the derivative and average of the delay times seen in Figure 4.2.

minutes, 30 minutes, and one hour. These jumps correspond to requests that are driven by fixed-interval timers.

The jumps can be observed more clearly if we take the derivative with respect to the y-axis of the cumulative distribution function. This can be seen in Figure 4.2b. The derivative is plotted along with its running average multiplied by 0.8. The average helps to illustrate places where the derivative drops below the amount of a typical fluctuation. The dips in the derivative that drop below the dotted line correspond to jumps in the distribution at times 30 seconds, 60 seconds, 90 seconds, 4 minutes, 5 minutes, 15 minutes, 30 minutes, 60 minutes respectively. Equations for the derivative and the average can be seen in Figure 4.3. *V* is a vector of delay times taken from every *n*th element in the full delay vector for a site. We chose the maximum of the square root of the full vector size or five for *n*. The value *a* represents the number of values used in the running average. We picked the maximum of the square root of the size of *V* or 3 for *a*.

4.3.3 Time of Day

The time of day during which a computer generates HTTP requests can help determine whether or not those requests are the result of human activity or come from an automated program. The optimal way to identify automated requests is to have an out-of-band input that tells us whether or not the user is actively using the computer. This could be achieved using a hardware device that passively monitors disk and keyboard activity, or by installing a monitoring program on each computer. However, direct information about whether the user is active may not always be available, especially with a passive network monitoring system.

المنارة للاستشارات
www.manaraa.com

```
Hour:       | 1                     24
Mar 26(Fri) | ---------------XXXXXXXXX
Mar 27(Sat) | --X---------------------
Mar 28(Sun) | ------------------------
Mar 29(Mon) | ---------------XX--XXX-X
Mar 30(Tue) | X--XX----------XXXXXXXXX
Mar 31(Wed) | ---------------XXXXXXXX-
```

Figure 4.4. Activity by time of day for one randomly chosen user. 12 AM to 1 AM is on the left, and 11 PM to 12 AM is on the right.

For cases where direct information about user activity is unavailable, we can analyze human usage patterns during a training period and build activity profiles. We found that people tend to follow fixed schedules. Figure 4.4 illustrates regular web browsing by one home user during the first six days of observation. The activity times stay fairly consistent from day to day. After a profile has been built for each user during an initial training period, we can mark HTTP requests made outside of typical usage times as likely having come from an automated network application rather than human browsing.

In our study, we looked at request timing for home users, many of whom were college students with irregular schedules. Still, we observed strikingly consistent browsing patterns. We expect the time-of-day approach to be even more effective in a work environment where employees have very regular schedules. Furthermore, the analysis could be extended to create special schedules for weekends and holidays when human browsing is much less likely in a work environment and more likely at home.

4.3.4 Eliminating False Positives from Refreshing Web Pages

Without any special processing, false positives will occur for web pages that periodically refresh. If a user leaves a refreshing web page open, its requests will trigger the time-of-day, inter-request delay, and request regularity filters. These types of refreshing pages are prevalent on the Internet and must be eliminated for consideration by the algorithms presented earlier in this section.

The way that we discount refreshing pages is by explicitly searching for refresh constructs within each web page. Refresh constructs include the HTML "`<meta http-equiv="refresh"…>`" tag as well as the Javascript "`setInterval`" and "`setTimeout`" functions. If a document contains a refresh construct, it is marked as *refreshing*. Requests for refreshing documents are treated as if they occurred at the time of the original document request. So, if a user loads a refreshing website before going home and leaves it open until the next day, then the time-of-day, regularity, and delay-time filters will not generate false positives.

www.manaraa.com

Additionally, we should ignore request times for resources that a client retrieves as a result reloading a refreshing page. If we only ignore request times for the refreshing page itself, then we will still see false positives for its embedded images and other objects. The best way of doing this is to examine the "referer" HTTP request header and determine the page that linked to the current object. If the referring page was loaded recently, then we can use its effective request time, which will be the first load time for a refreshing page. This way, we can avoid false positives from image and object loads associated with refreshing pages.

4.4 Formatting Analysis

The HTTP protocol specification allows for a wide range of message types and header fields. However, the set of possible HTTP requests that a web browser may send at any given time is much more limited. The goal of our formatting analysis is to determine the set of HTTP requests $\boldsymbol{R}$ that a web browser may send and mark any other requests $\boldsymbol{r} \notin \boldsymbol{R}$ as coming from a non-browser network application. These flagged requests are subsequently fed through a whitelist to determine their source network application. The whitelisting process is discussed in greater detail in chapter 6.

There are two general strategies for enumerating the set $\boldsymbol{R}$ of HTTP requests that a browser may send: *stateful* analysis and *stateless* analysis. The result of stateless analysis is a language of all possible network outputs that a browser would ever generate, regardless of its inputs. Stateless analysis is simple and yields a large domain of potential HTTP requests. Stateful analysis, on the other hand, takes previous program inputs into account when determining the set of possible HTTP requests. A network monitoring system only has network inputs available to it, but stateful analysis running on the same computer as the web browser could also consider inputs such as mouse clicks, key presses, and clipboard paste operations.

المنارة للاستشارات
www.manaraa.com

```
GET /search?hl=en&q=security&btnG=Google+Search HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.12)
       Gecko/20080201 Firefox/2.0.0.12
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,
       text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.google.com/
Cookie: PREF=ID=0a2154dbc55bdbef:TM=1206244622:LM=1206244622:S=Ef4gW8gUXozO0f
```

Figure 4.5. A sample HTTP GET request sent to Google.com by the Firefox browser (version 2.0.0.12) following a search for the term “security.” The “User-Agent” header field contains a compound value identifying the operating system (Windows XP), the language (en-US), and the browser.

HTTP requests consist of three sections: the request line, the request headers, and the request body. A sample HTTP request for the search term “security” at www.google.com can be seen in Figure 4.5. The first line of the HTTP request is the *request line*. The first word on this line is the request method, which is usually either “GET” for obtaining a resource or “POST” for submitting data in almost all cases. Next is the path of the requested resource, which forms the request URL when combined with the hostname. The last part of the request line is the HTTP version, which is either 1.0 or 1.1 for any modern web client. Because the request method and HTTP version have so few possible values, they are rarely helpful in identifying network applications. In a stateless formatting analysis, the path can take on any value that is a valid URL string for a legitimate browser, so it also cannot help us identify network applications.

We focus on the header fields following the request line to uncover information about the client that made the request. The HTTP specification describes the use of numerous standard header fields, but allows for custom *extension header fields* [Fielding99]. An extension header field can be any alphabetic string (optionally including the characters ‘-’ and ‘_’). The presence of particular extension header fields will often uniquely identify a network application. We flag any HTTP request that contains header fields not present in requests made by standard web browsers.

www.manaraa.com

Filter Name		# Alerts	Avg. Alerts / Day	False Positives (Percentage)
Message Format		240	6.00	0 (0%)
Delay Time		118	2.95	6 (5%)
Request Regularity	8-Hour	132	3.30	15 (11%)
	48-hour	65	1.63	5 (8%)
Time of Day		68	2.62	19 (28%)
Aggregate		623	16.5	45 (7%) (avg. 1.13 alerts / day)

Table 4.1. Number of alerts and the percentage of alerts that are false positives for each filter. The aggregate row shows results from running all the filters in parallel.

The HTTP specification includes an optional standard header field known as the "User-Agent." The purpose of the User-Agent field is for clients to explicitly identify themselves. Legitimate network applications will usually include a unique string in the User-Agent field. Only a few programs, such as news readers, will omit the field entirely. Standard web browsers use special compound User-Agent values that may include information about the client operating system, browser plug-ins, client language, etc. An example of a compound User-Agent value can be seen in Figure 4.5. Because each part of a compound User-Agent may be associated with a different client application component, we split User-Agent values in this format and flag requests with elements that are not present in standard browser requests. This not only helps in identifying legitimate network applications, but also exposes certain adware and spyware programs that masquerade as useful browser plug-ins.

4.5 Traffic Evaluation

After a one-week learning period, during which we designed the filters and set their thresholds, we put the filters to the test against 40 days of web traffic from 30 users. The 40 days of web traffic included 428,608 requests to 6441 different websites totaling 300 Megabytes in size. During the evaluation, all the filters were active for every site and user. The purpose was to measure how effective the filters were at differentiating automated web activity from human browsing, including the false positive rate. We did not apply any whitelist rules to the resulting alerts to classify their source network application. To determine false positives, we only checked to see whether each alert was, in fact, the result of non-browser network application activity.

Table 4.1 summarizes the results of the evaluation. A total of 623 alerts were generated over the 40-day evaluation period for 30 users, an average of 0.55 alerts per user per day. These alerts led to the identification of seventeen different non-browser network applications and one non-standard browser. Six of these were unwanted spyware programs. We found that at least 5

المنارة للاستشارات
www.manaraa.com

out of the 30 observed users had some form of adware or spyware on their computers. In addition to the spyware programs, others were detected that may not be desirable in a work environment. These included Kazaa, iTunes, AIM Express, and BitTorrent. Benign network applications that we were able to identify include Windows Update, McAfee Web Update, and others.

During the evaluation, there were only 45 false positives total, an average of 1.13 per day for all 30 users. In a network with 1000 computers, this extrapolates to about 38 false positives per day. Keep in mind, however, that this traffic is from home computers with diverse usage patterns. Some of the false positives that arose, such as those from continually browsing a social networking site for two hours, should be less likely in an enterprise environment. Furthermore, we could eliminate false positives from the time-of-day filter, which generated the most false positives, by incorporating out-of-band information about peoples' schedules or by getting rid of the filter altogether. Not counting the time-of-day filter, there was an average of 0.65 false positives per day. Considering these factors, we believe the number of false positives to be reasonable for enterprise deployment.

4.5.1 Regularity

The regularity filter results seen in Table 4.1 consisted of both count and coefficient of variation measurements. We considered the number of false positives generated by this filter to be acceptable (approximately one false alarm every three days). The servers that caused false alarms hosted popular websites such as ebay.com and livejournal.com. Many of the sites flagged by the regularity filter were found by the delay time filter as well. The regularity filter did, however, find an additional type of spyware that the delay filter was unable to detect: browser search bars. This particular breed of unwanted program embeds itself into the person's browser and calls back to its host every time the browser opens up, as well as throughout the browsing session. These are different from other malware programs because their callbacks are triggered by human activity and thus cannot easily be differentiated from a person based on inter-request delay times. We successfully detected sites that used frequent requests with this filter, even if they coincided with human usage.

4.5.2 Inter-Request Delay

For the delay time measurements, we logged website access times using one-second granularity. The reason we did not use more precision is that none of the timers observed had periods of less than 30 seconds. In order to detect shorter-period timers, additional precision would be required to differentiate a timer from repeated short delay times. The false positive rate

المنارة للاستشارات
www.manaraa.com

for the delay time filter was low (an average of one false alarm every 6 days for our test group). These false positives came from websites whose refresh mechanisms we were not able to detect with our false positive reduction algorithm.

4.5.3 Time of Day

The time of day filter was initially configured using the one-week training period. After seeing preliminary results, we lengthened the training time to also include the first week of the 40-day period so it was two weeks total. This increased the effectiveness of the filter, as it may take a few weeks to accurately capture browsing patterns. It is important to note that some automated network applications *were* active during the training period. We did not attempt to remove non-human activity from the training data. The effectiveness of training could be improved to generate more true positives by removing traffic for sites that are identified by the other filters as receiving traffic from automated network applications. Nevertheless, we were able to detect programs such as Gator and Wildtangent even though they had been active during the training period. This may have been caused by post-training installation, or by changes to the schedule of when a computer is *on*, but not actively used.

4.5.4 Formatting

The large number of formatting alerts can be attributed to the fact that the formatting filter raises an alarm when it sees a bad header once for each web server per user. This means that if iTunes were to access 10 different sites, each would generate an alarm. The whitelisting techniques we present in chapter 6 help aggregate these duplicate alerts for known network applications.

4.6 HTTP Tunnel Evaluation

We tested the effectiveness of the timing and formatting filters against a number of HTTP tunnel and backdoor programs. These programs are designed to blend automated activity in with legitimate web traffic to bypass firewalls and avoid detection. The tunneling programs that we tested include Wsh [Dyatlov09a], Hopster [Hopster09], and Firepass [Dyatlov09b]. We also tested a backdoor program that we designed, Tunl which allows a hacker outside the network to remotely control a machine behind a firewall using a command-shell interface and HTTP request callbacks.

المنارة للاستشارات
www.manaraa.com

4.6.1 Third Party HTTP Tunnels

We installed the three tunneling programs on a computer and sent out information using each. The format filter was immediately able to detect both Wsh and Firepass because they used custom header fields in their requests. Following its initial connection, we were unable to successfully transfer any data using Firepass. Wsh did work properly, but did not trigger any timing alerts because it generated requests in response to human input.

We used Hopster to tunnel traffic from AOL Instant Messenger in our experiments. It began running at 10:30 PM and no messages were sent during the night. The next day, 10 KB of data was sent out around Noon. Hopster was not detected immediately like Firepass and Wsh because it copied web browser request formatting. Unlike the other two programs, Hopster did make frequent callbacks to its server that triggered the regularity filter after 80 minutes and the delay time filter after two hours.

4.6.2 Tunl Design

To further evaluate our system, we also designed a prototype remote shell backdoor called *Tunl*. It is made to simulate the scenario where a hacker is controlling a compromised computer that is behind a firewall with a remote command shell interface. Tunl consists of two executables, a client, *TunlCli*, that runs on the compromised host, and a server, *TunlServ*, that runs on run on a machine controlled by the attacker. Tunl can tunnel its traffic through an HTTP proxy server or send its HTTP requests directly to the Internet, blending in with normal web traffic.

The first thing TunlCli does when it starts up is launch a hidden command shell with redefined standard input, output, and error handles. It then redirects the input and output from the command shell to a remote console running on TunlServ using HTTP requests. In addition to forwarding data from the command shell output, it makes periodic callbacks to check for server commands. Custom get and put commands, which are not piped to the shell, are included in Tunl for easy file transfers. To avoid sending too many small requests, data is buffered and sent out every 100 milliseconds.

Although the attacker has an illusion of a command shell on the Tunl server, requests may take a long time to execute because they are fetched by periodic TunlCli callbacks. The server has no way of directly connecting to the client. It has to wait for a ping in the form of an HTTP request, and then return commands in the body of an HTTP reply. Callbacks were scheduled at one-hour intervals, with two optional retries at 30-second intervals following each callback for failed connection attempts. Only calling back every hour ensures that Tunl generates

المنارة للاستشارات
www.manaraa.com

a low volume of HTTP requests and blends in with normal traffic. All of the messages exchanged between the client and server match the format of an Internet Explorer web browser and a standard-configuration Apache web server, respectively. This avoids detection by formatting filters.

4.6.3 Tunl with Callback-Only Workload

To evaluate the performance of timing filters in this chapter, we installed the Tunl program and monitored its traffic. The first workload we tested consisted only of callbacks to the Tunl server (the Tunl client and server are connected, but the server did not issue commands). This represents the time when a machine has compromised but is not actively executing commands. The results for the Tunl client only making callbacks were promising. Even though the client executed no commands, the traffic from this trace was caught by the request regularity, delay time, and time-of-day filters. The 8-hour coefficient of variation filter detected the web tunnel 6 hours and 40 minutes after the first callback. The 8-hour activity count filter was unable to detect the backdoor. Tunl did, however, break the threshold for the 48-hour count filter after about 26 hours. Since the backdoor was running on a timer, the delay time filter was able to detect it in 2 hours and 10 minutes. As far as the time of day filter, the delay until detection varies depending on the individual user’s habits as well as the time of initial callback. The time of day filter was triggered by the backdoor very shortly after a time of usual inactivity began.

4.6.4 Minimal Workload

The second test case consisted of a hacker using the Tunl shell to go to the local documents directory (containing approximately 180 documents), listing all the files, and downloading a single 500-word uncompressed document with minimal formatting (approximately 25 KB). This is a minimal activity scenario where an attacker only lists one directory and downloads a single small file. This workload triggered the delay time and request regularity filters. In the presence of more concentrated activity associated with the file transfer, however, the backdoor was harder to detect using the coefficient of variation regularity measurement. Instead of detecting Tunl in around 7 hours, the coefficient of variation measurement did not pass the threshold until after the file transfer activity was beyond the 8-hour measurement window.

المنارة للاستشارات
www.manaraa.com

4.6.5 Moderate Workload

The third test case involved a moderately intensive remote shell session. We listed all local document and desktop directories for one user on the machine. Following the directory list requests, we compressed and downloaded a variety of files including two income tax returns (PDF format), one JPG image, three small Word documents, and a text file containing a 1000-address mailing list. The moderate workload generated the same alerts as the minimal workload in the same amount of time. The moderate workload did take longer than the minimal workload to complete, but the difference was between two minutes and ten minutes of transfer activity, which was too short to have any noticeable effect on the 5-minute-granulatiy regularity measurements.

4.7 Filter Vulnerabilities

Although the filters presented in this chapter are very effective at identifying non-browser network applications, it is still possible to avoid them and impersonate human web browsing. For each type of filter, here are steps that malware could take to evade detection:

- *Delay Time Filter* – Randomize callbacks so as to not exceed thresholds (though this can still trip time-of-day filter, if the user is not usually active at that time.)
- *Time-of-day Filter* – Schedule requests when a user is normally active by monitoring user activity (though this increases the risk of detection by the user).
- *Request Regularity Filter* – If the thresholds are known, this filter can be avoided by computing regularity and staying below them. In general, constraining regularity to that of a typical legitimate website will avoid detection, but it will also restrict the amount of time during which a malicious network application can communicate with its host.
- *Message Formatting* – This filter is much easier to avoid; one only has to mimic the formatting of requests from the web browser installed on the compromised machine. However, using the right browser version is important. If malware mimics a browser that is not installed on any computer in an enterprise network, it may still be detected by message formatting analysis

Despite these filter vulnerabilities, we believe that the filters presented in this chapter significantly raise the bar for malicious software that wishes to avoid detection by blending in with normal web traffic. Furthermore, these algorithms will effectively identify legitimate non-browser network applications and unwanted applications, such as file sharing programs, which do not actively try to escape detection.

المنارة للاستشارات
www.manaraa.com

4.8 Conclusion and Future Work

In this chapter, we demonstrated methods for differentiating automated network application traffic from normal browsing. These methods are based on observations from real traffic from 30 users during a one-week training period. The first set of techniques focus on request timing characteristics. The delay between requests, their regularity, and the time of day at which they occur serve as a good differentiator between human and automated activity. We also presented methods for classifying non-browser network application traffic by looking at the formatting of requests. In particular, customized header fields and the client-specified "User-Agent" field can uniquely identify many network applications.

For our evaluation, we ran the resulting timing and formatting filters on real traffic from 30 users over a 40-day period. The filters were effective in identifying traffic from seventeen non-browser network applications, six of which were spyware. They also had a low false positive rate of approximately one per day for the duration of the evaluation. We also tested the filters against applications known as HTTP tunnels that are specifically designed to avoid detection and bypass firewalls. The formatting filters immediately detected two of three publicly available HTTP tunnels, and timing filters detected the third after 80 minutes. We also created and tested a custom HTTP tunnel that was quieter than publicly available tunnel programs. The timing algorithms were able to identify this custom tunnel after approximately two hours.

In the future, we hope to extend the formatting analysis techniques to take the current browsing session state into account. The formatting filters we present here look at all requests regardless of what pages the client has loaded in the past. If we also take browsing session state into account, then we can flag requests that exhibit inconsistencies with normal browser behavior and may be malicious (e.g., submitting a POST request to a URL to which there are no links from previous pages). This would require browser session state tracking and in-depth analysis of HTML and Javascript in server replies. However, it would greatly enhance security by preventing spyware from directly leaking information to any server that does not have a link from a currently open page.

المنارة للاستشارات
www.manaraa.com

CHAPTER 5

QUANTIFYING INFORMATION LEAKS IN OUTBOUND WEB TRAFFIC

5.1 Overview

Network-based information leaks pose a serious threat to confidentiality. They are the primary means by which hackers extract data from compromised computers. The network can also serve as an avenue for insider leaks, which, according to a 2007 CSI/FBI survey, are the most prevalent security threat for organizations [Richardson07]. Because the volume of legitimate network traffic is so large, it is easy for attackers to blend in with normal activity, making leak prevention difficult. In one experiment, a single computer that was browsing a social networking site for 30 minutes generated over 1.3 MB of legitimate request data—the equivalent of about 195,000 credit card numbers. Furthermore, manually analyzing network traffic for leaks would be unreasonably expensive and error-prone. Due to the heavy volume of normal traffic, limiting network traffic based on the raw byte count would only help stop large information leaks.

In response to the threat of network-based information leaks, researchers have developed data-loss prevention (DLP) systems [RSA07, Vontu09]. DLP systems work by searching through outbound network traffic for known sensitive information, such as credit card and social security numbers. Some even catalog sensitive documents and look for excerpts in outbound traffic. Although they are effective at stopping accidental and plain-text leaks, DLP systems are fundamentally unable to detect obfuscated information flows. They leave an open channel for leaking data to the Internet.

This chapter introduces a new approach for precisely quantifying information leak capacity in network traffic. Rather than searching for known sensitive data—an impossible task in the general case—we aim to measure and constrain its maximum volume. This research addresses the threat of a hacker or malicious insider extracting sensitive information from a network. He or she could try to steal data without being detected by hiding it in the noise of normal outbound

المنارة للاستشارات
www.manaraa.com

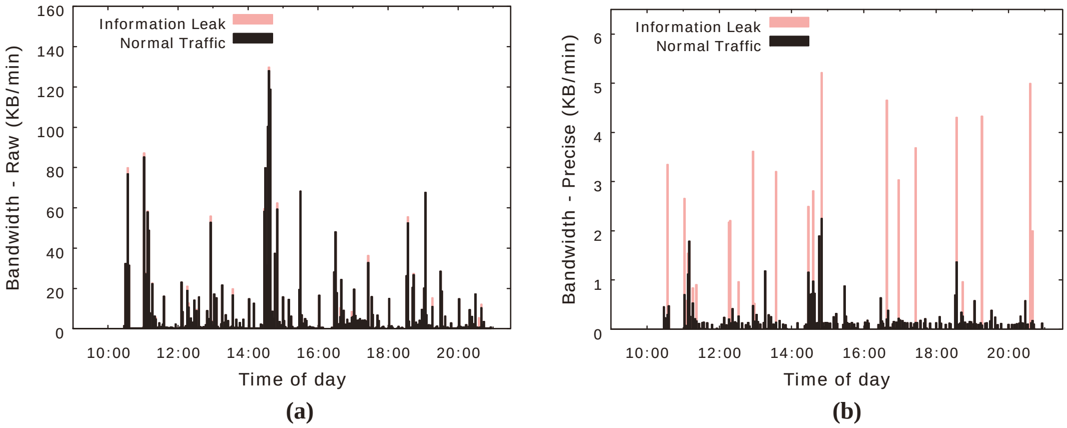


Figure 5.1. Graph of outbound web traffic during a typical work day with a 100 Kilobyte information leak inserted. (a) shows the raw byte count, where the leak is barely noticeable, and (b) shows the precise unconstrained bandwidth measurement, in which the leak stands out prominently.

traffic. For web traffic, this often means stashing bytes in paths or header fields within seemingly benign requests. To combat this threat, we exploit the fact that a large portion of legitimate network traffic is repeated or constrained by protocol specifications. This fixed data can be ignored, which isolates real information leaving a network, regardless of data hiding techniques.

The leak measurement techniques presented here focus on the Hypertext Transfer Protocol (HTTP), the main protocol for web browsing. They take advantage of HTTP and its interaction with Hypertext Markup Language (HTML) documents and Javascript code to quantify information leak capacity. The basic idea is to compute the expected content of HTTP requests using only externally available information, including previous network requests, previous server responses, and protocol specifications. From the perspective of information theory, this expected content is equivalent to the signal distribution. The amount of *unconstrained* outbound bandwidth, which is equivalent to background entropy, is equal to the edit distance (edit distance is the size of the edit list required to transform one string into another) between actual and expected requests, plus timing information. Given correct assumptions about timing channel characteristics, these results may overestimate, but will never underestimate the true size of information leaks, thus serving as a tight upper bound on information leakage.

One option for measuring unconstrained bandwidth would be to use a traditional compression algorithm like gzip [Gailly08], bzip2 [Seward07], or lzma [Pavlov09]. This would involve building up a library from previous messages and only counting the incremental size of new requests. Traditional compression can help for simple requests that have large repeated substrings. However, this protocol-agnostic approach fails to capture complex interactions between requests and replies that go beyond string repetition.

المنارة للاستشارات
www.manaraa.com

The analysis techniques presented in this chapter take advantage of protocol interactions. Parsing all of the links on a web page, for example, helps construct an accurate distribution of expected requests. Our analysis also involves executing scripts in a simulated browser environment to extract links that cannot be derived from static processing. These improvements lead to a much more precise measurement of information in outbound web traffic than conventional compression algorithms.

Figure 5.1 illustrates the benefit of precise leak quantification. The graphs show bandwidth from legitimate web browsing over a one-day period in black. A 100 KB information leak was inserted into the traffic and can be seen in a lighter color. This leak was deliberately inserted in short bursts, so as to more closely resemble legitimate web traffic and avoid detection methods that look at request regularity, which are discussed in Chapter 4. The left graph shows raw request bandwidth. The leak is barely noticeable here and easily blends in with the noise of normal activity. After running the same traffic through our unconstrained bandwidth measurement engine, however, the leak stands out dramatically from normal traffic. It is important to note that more accurate traffic measurement does not completely stop information leaks from slipping by undetected; it only makes it possible to identify smaller leaks. Our analysis techniques force a leak that would normally blend in with a week's worth of traffic to be spread out over an entire year.

We evaluated our leak measurement techniques on real browsing data from 10 users over 30 days, which included over 500,000 requests. The results were compared to a simple calculation described in prior research [Borders04], and to incremental gzip compression [Gailly08]. The average request size using the leak measurement techniques described in this chapter was 15.8 bytes, 1.5% of the raw byte count. The average size for gzip was 132 bytes, and for the simple measurement was 243 bytes. The experiments show that our approach is an order of magnitude better than traditional gzip compression.

This work focuses specifically on analyzing leaks in HTTP traffic for a few reasons. First, it is the primary protocol for web browsing and accounts for a large portion of overall traffic. Many networks, particularly those in which confidentiality is a high priority, will only allow outbound HTTP traffic and block everything else by forcing all traffic to go through a proxy server. In this scenario, HTTP would be the only option for directly leaking data. Another reason for focusing on HTTP is that a high percentage of its request data can be filtered out by eliminating repeated and constrained values.

The principles we use to measure leaks in HTTP traffic are likely to work for other protocols as well. Binary protocols for instant messaging, secure shell access, and domain name

للاستشارات المنارة
www.manaraa.com

resolution all contain a number of fixed and repeated values. Furthermore, correlation between protocols may enable filtering of DNS lookups. Extending a similar methodology to outbound SMTP (e-mail) traffic is likely to be more challenging. E-mail primarily consists of free-form data and only contains small fixed fields. However, the unconstrained data in e-mails is usually text, for which there are well-known methods of determining the information content [Shannon51], or file attachments. These attachments are made up of data written out in a specific file format, which could be analyzed in a manner similar to HTTP. In fact, researchers have already examined ways of identifying information that has been hidden in files with steganography by looking for additional unexpected entropy [Anderson98]. Further investigation of leak measurement techniques for file attachments and other protocols is future work.

The measurement techniques in this chapter do not provide an unconstrained bandwidth measurement for *fully* encrypted traffic. (If a hacker tries to hide or tunnel encrypted data in an unencrypted protocol, it can be measured.) All networks that allow outbound encrypted traffic must deal with this fundamental problem, and we do not try to solve it here. If confidentiality is a top priority, there are a few possibilities for obtaining original plain text. One is to force all encrypted traffic through a gateway that acts as a man-in-the-middle on each connection. This can be achieved by designating the gateway as a local certification authority and having it rewrite certificates. Another option is to deploy an agent on every host in the network that reports encryption keys to a monitoring system. With this approach, any connections that cannot be decrypted are subsequently blocked or flagged for further investigation.

The leak measurement techniques presented in this chapter do not constitute an entire security solution, but rather act as a tool. We envision the primary application of this work to be forensic analysis. One could filter out almost all legitimate activity, making it faster and easier to isolate leaks. Another application would be intrusion detection. Additional research would be required to determine appropriate thresholds and optimize the algorithms for handling large volumes of traffic.

The remainder of this chapter is laid out as follows. Section 5.2 discusses related work. Section 5.3 poses a formal problem description. Section 5.4 talks about static message analysis techniques. Section 5.5 describes dynamic content analysis methodology. Section 5.6 outlines an approach for quantifying timing information. Section 5.7 presents evaluation results. Section 5.8 discusses potential strategies for mitigating entropy and improving analysis results. Finally, section 5.9 concludes and suggests future research directions.

للاستشارات المنارة

www.manaraa.com

5.2 Related Work

There are numerous techniques for controlling information flow within a program. Jif [Myers01] ensures that programs do not leak information to low-security outputs by tainting values with sensitive data. More recent work by McCamant et al. [McCamant08] goes one step further by quantifying amount of sensitive data that each value in a program can contain. Unfortunately, intra-program flow control systems rely on access to source code, which is not always feasible. They do not protect against compromised systems. The algorithms in this chapter take a black box approach to measuring leaks that makes no assumptions about software integrity.

Research on limiting the capacity of channels for information leakage has traditionally been done assuming that systems deploy mandatory access control (MAC) policies [Brand85] to restrict information flow. However, mandatory access control systems are rarely deployed because of their usability and management overhead, yet organizations still have a strong interest in protecting confidential information.

A more recent system for controlling information flow, TightLip [Yumerefendi07], tries to stop programs from leaking sensitive data by executing a shadow process that does not see sensitive data. Outputs that are the same as those of the shadow process are treated normally, and those that are different are marked confidential. TightLip is limited in that it relies on a trusted operating system, and only protects sensitive data in files. In comparison, our leak measurement methods will help identify leaks from a totally compromised computer, regardless of their origin.

A popular approach for protecting against network-based information leaks is to limit where hosts can send data with a content filter, such as Websense [Websense09]. Content filters may help in some cases, but they do not prevent all information leaks. A smart attacker can post sensitive information on *any* website that receives input and displays it to other clients, including useful sites such as www.wikipedia.org. We consider content filters to be complimentary to our measurement methods, as they reduce but do not eliminate information leaks.

Though little work has been done on quantifying network-based information leaks, there has been a great deal of research on methods for leaking data. Prior work on convert network channels includes embedding data in IP fields [Cabuk04], TCP fields [Servetto01], and HTTP protocol headers [Castro06]. The methods presented in this chapter aim to quantify the maximum amount of information that an HTTP channel could contain, regardless of the particular data hiding scheme employed.

Other research aims to reduce the capacity of network covert channels by modifying packets. Network “pumps” [Kang95] and timing jammers [Giles03] control transmission time to

المنارة للاستشارات
www.manaraa.com

combat covert timing channels. Traffic normalizers (also known as protocol scrubbers) will change IP packets in flight so that they match a normal format [Handley01, Malan00]. Glavlit is an application-layer protocol scrubber that focuses specifically on normalizing HTTP traffic from servers [Schear06]. Traffic normalization helps eliminate covert storage channels by fixing ambiguities in network traffic. Research on normalizing network traffic to reduce covert channel capacity is complimentary to our work, which focuses only on quantifying information content.

Compression algorithms have been the subject of much research in the information theory community. Popular compression algorithms, such as gzip [Gailly08], bzip2 [Seward07], and lzma [Pavlov09], combine techniques from prior research to achieve fast and efficient compression. Gzip and bzip2 use Huffman coding to exploit redundancy at the byte level [Huffman52]. All three of the popular compression algorithms use a variant of the algorithms introduced by Lempel and Ziv for compressing repeated data strings [Ziv77]. The lzma algorithm, which usually has the best compression ratio [Morse05], also applies a “Range” algorithm, which is similar to arithmetic coding [Rissanen79]. Despite improvements from more clever compression methods, non of these algorithms can achieve the same level of compression as the techniques presented in this chapter, because they do not have knowledge of HTTP, HTML, and Javascript. Complex interactions on web pages—including the execution of code—are not exploited by generic compression algorithms.

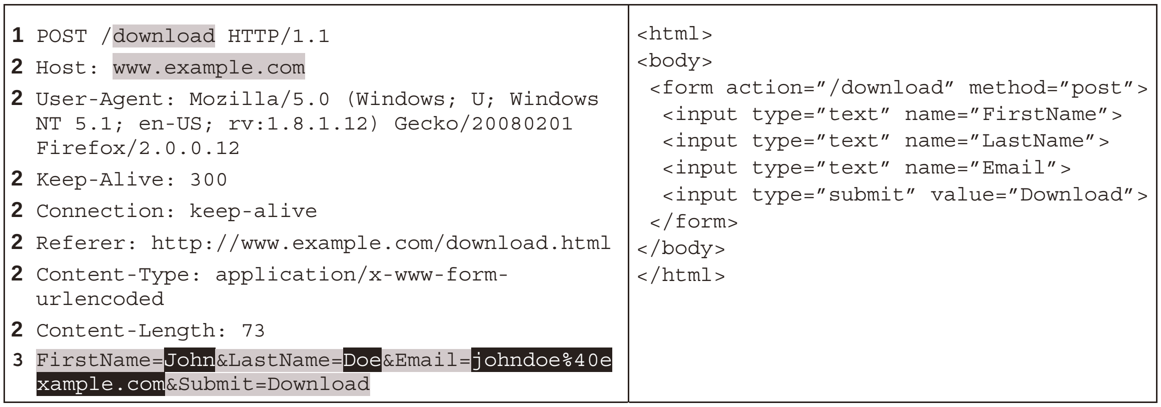
5.3 Problem Description

This chapter addresses the problem of quantifying network-based information leak capacity by isolating information from the client in network traffic. We will refer to information originating from the client as *UI-layer* input. From a formal perspective, the problem can be broken down to quantifying the set ***U*** of UI-layer input to a network application given the following information:

- ***I*** **–** The set of previous network inputs to an application.
- ***O*** **–** The set of current and previous network outputs from an application.
- ***A*** **–** The application representation, which is a mapping: $\boldsymbol{U} \times \boldsymbol{I} \rightarrow \boldsymbol{O}$ of UI-layer information combined with network input to yield network output.

By definition, the set ***I*** cannot contain new information from the client because it is generated by the server. In this work, the application representation ***A*** is based on protocol specifications, but it could also be derived from program analysis. In either case, it does not contain information from the client. Therefore, the information content of set ***O*** can be reduced to

المنارة للاستشارات
www.manaraa.com



```
1 POST /download HTTP/1.1
2 Host: www.example.com
2 User-Agent: Mozilla/5.0 (Windows; U; Windows
  NT 5.1; en-US; rv:1.8.1.12) Gecko/20080201
  Firefox/2.0.0.12
2 Keep-Alive: 300
2 Connection: keep-alive
2 Referer: http://www.example.com/download.html
2 Content-Type: application/x-www-form-
  urlencoded
2 Content-Length: 73
3 FirstName=John&LastName=Doe&Email=johndoe%40e
  xample.com&Submit=Download
```

(a)

```
<html>
<body>
 <form action="/download" method="post">
  <input type="text" name="FirstName">
  <input type="text" name="LastName">
  <input type="text" name="Email">
  <input type="submit" value="Download">
 </form>
</body>
</html>
```

(b)

Figure 5.2 (a) A sample HTTP POST request for submitting contact information to download a file. Line **1** is the HTTP request line. Lines marked **2** are request headers, and line **3** is the request body. Bytes counted by a simple algorithm are highlighted in gray. UI-layer data is highlighted in black with white text. (b) A sample HTML document at http://www.example.com/download.html that generated request (a).

the information in the set ***U***. If the application has been tampered with by malicious software yielding a different representation ***A'***, then the maximum information content of tampered output ***O'*** is equal to the information content of the closest expected output ***O*** plus the edit distance between ***O*** and ***O'***. Input supplied to an application from all sources other than the network is considered part of ***U***. This includes file uploads and system information, such as values from the random number generator. Timing information is also part of the set ***U***.

5.4 Static Content Analysis

This section describes methods for measuring the amount of information in outbound HTTP requests by statically analyzing previous requests and responses. Some portions of the request headers are fixed and can be immediately filtered if they contain the expected values. Most of the header fields only change on rare occasion and can be discounted if they are the same as previous requests. The request path, which identifies resources on the web, is usually derived from previous HTML pages returned by the server. Filtering out repeated path values requires comparing paths to those in both prior requests and responses. Also, HTTP form post requests reference field names and default values contained in HTML pages. This section elaborates on methods for extracting expected HTTP request fields from static content.

www.manaraa.com

5.4.1 HTTP Request Overview

There are two main types of HTTP requests used by web browsers, GET and POST. GET typically obtains resources and POST sends data to a server. An example of a HTTP POST request can be seen in Figure 5.2. This request is comprised of three distinct sections: the request line, headers, and the request body. GET requests are very similar except that they do not have a request body. The request line contains the path of the requested file on the server, and it may also have script parameters. The next part of the HTTP request is the header field section, which consists of "<*field*>: <*value*>" pairs separated by line breaks. Header fields relay information such as the browser version, preferred language, and cookies. Finally, the HTTP request body comes last and may consist of arbitrary data. In the example message, the body contains an encoded name and e-mail address that was entered into a form.

5.4.2 HTTP Header Fields

The first type of HTTP header field that we examine is a fixed header field. Fixed headers should be the same for each request in most cases. Examples include the preferred language and the browser version. We only count the size of these headers for the first request from each client, and count the edit distance from the most recent request on subsequent changes. Here, we treat all HTTP headers except for Host, Referer, and Cookie as fixed. Some of these header fields, such as Authorization, may actually contain information from the user. When these fields contain new data, we again count the edit distance with respect to the most recent request.

Next, we look at the Host and Referer header fields. The Host field, along with the request path, specifies the request's uniform resource locator (URL). We only count the size of the Host field if the request URL did not come from a link in another page. Similarly, we only count the Referer field's size if does not contain the URL of a previous request.

Finally, we examine the Cookie header field to verify its consistency with expected browser behavior. The Cookie field is supposed to contain key-value pairs from previous server responses. Cookies should never contain UI-layer information from the client. If the Cookie differs from its expected value or we do not have a record from a previous response (this could happen if a mobile computer is brought into an enterprise network, for example), then we count the edit distance between the expected and actual cookie values. At least one known tunneling program, Cooking Channel [Castro06], deliberately hides information inside of the Cookie header in violation of standard browser behavior. The techniques presented here correctly measure outbound bandwidth for the Cooking Channel program.

المنارة للاستشارات
www.manaraa.com

5.4.3 Standard GET Requests

HTTP GET requests are normally used to retrieve resources from a web server. Each GET request identifies a resource by a URL that is comprised of the server host name, stored in the Hostname header field, and the resource path, stored in the request line. Looking at each HTTP request independently, one cannot determine whether the URL contains UI-layer information or is the result of previous network input (i.e., a link from another page). If we consider the entire browsing session, however, then we can discount request URLs that have been seen in previous server responses, thus significantly improving unconstrained bandwidth measurements.

The first step in accurately measuring UI-layer information in request URLs is enumerating all of the links on each web page. We parse HTML, Cascading Style Sheet (CSS), and Javascript files to discover static link URLs, which can occur in many different forms. Links that are written out dynamically by Javascript are covered in section 5.5. Examples of static HTML links include:

- `<a href=“http://test.com/page”>Click Here!</a>`
- `<link rel=stylesheet type=”text/css” href=”style.css”>`
- `<img src=”image.jpg”>`
- And the less common: `<script src=”//test.com/preload.jpg”>`

These examples would cause the browser to make requests for “page”, “style.css”, “image.jpg”, and “preload.jpg”, respectively.

After the set of links has been determined for each page, we can measure the amount of UI-layer information conveyed by GET requests for those link URLs. The first step is identifying the link’s referring page. HTTP requests typically identify the referrer in a header field. If the referrer is found, then the request URL is compared against a library of *mandatory* and *voluntary* links on the referring page. Mandatory links are those that should always be loaded unless they are cached by the browser, such as images and scripts. The set of mandatory links is usually smaller and more frequently loaded. Voluntary links are those that the browser will not load unless the user takes some action, such as clicking on a link. Voluntary links tend to be more numerous and are loaded less often. Finally, if a request does not identify the referrer or the referring page cannot be found, then we must go to the library of *all* previously seen links to look for a match.

Once a matching link from one of the three groups (mandatory, voluntary, or all) has been found, the amount of information in the request is measured as the sum of:

- 2 bits to identify the link group

www.manaraa.com

- Log(*n*) bits to identify the link within the group, where *n* is the total number of links in the group
- The edit distance from the link URL to the actual request URL if it is not an exact match

For approximate matches, calculating the edit distance from all URLs would be prohibitively expensive. Instead we select only a few strings from which to compute the edit distance, and then take the best answer. This pre-selection is done by finding strings with the longest shared substring at the beginning. Our original plan for mandatory links was to not count any data if all the mandatory links were loaded in order. This works in a controlled environment, but our experiments showed that local caching prevents the browser from loading most of the mandatory links in many cases. A simpler and more effective approach is to independently count the link information in each request. This includes information conveyed by the client about whether it has each object in its cache.

5.4.4 Form Submission Requests

The primary method for transmitting information to a web server is form submission. Form submission requests send information that the user enters into input controls, such as text boxes and radio buttons. They may also include information originating from the server in hidden or read-only fields. Form submissions contain a sequence of delimited <*name, value*> pairs, which can be seen in the body of the sample POST request in Figure 5.2a. The field names, field ordering, and delimiters between fields can be derived from the page containing the form, which is shown in Figure 5.2b, and thus do not convey UI-layer information. Field values may also be taken from the encapsulating page in some circumstances. Check boxes and radio buttons can transmit up to one bit of information each, even though the value representing "on" can be several bytes. Servers can also store client-side state by setting data in "hidden" form fields, which are echoed back by the client upon form submission. Visible form fields may also have large default values, as is the case when editing a blog post or a social networking profile. For fields with default values, we measure the edit distance between the default and submitted values. We measure the full size of any unexpected form submissions or form fields, which may indicate an attempt to leak data.

5.5 Dynamic Content Analysis

Very few websites today are free from active content. This poses a challenge for leak measurement because such content may generate HTTP requests with variable URLs. The data in

المنارة للاستشارات
www.manaraa.com

these requests might still be free from UI-layer information, but making this determination requires dynamic content analysis. This section describes methodology for processing and extracting expected HTTP request URLs from active web content.

5.5.1 Javascript

The most popular language for dynamic web page interaction is Javascript, which is implemented by almost all modern browsers. Javascript has full access to client-side settings, such as the browser version and window size, which help it deliver the most appropriate content to the user. On many websites, Javascript will dynamically construct link URLs. These URLs cannot be extracted from simple parsing. One must execute the Javascript to obtain their true values.

The leak analysis engine includes a Javascript interpreter, SpiderMonkey [Mozilla09b], to handle dynamic link creation. When processing an HTML document, the analysis engine first extracts static links as described in the previous section, and then executes Javascript code. A large portion of links that Javascript generates are written out during the page load process. This includes tracking images, advertisements, embedded media content, and even other scripts. The analysis engine executes Javascript as it is encountered in the HTML document in the same way as a web browser. This includes complex chaining of script tags using both the "document.write('<script…')" method, and the "node.addChild(document.createElement('script'))" method. When scripts add HTML or DOM nodes to the document, the analysis engine processes the new document text, looking for newly created links. Executing scripts allows the engine to see a large set of links that are unrecoverable with static parsing.

5.5.2 The DOM Tree

Javascript is a stand-alone language that only has a few built-in types and objects. Most of the rich interface available to scripts inside of web pages is defined by the web browser as part of the Document Object Model (DOM). All of the elements in an HTML document are accessible to Javascript in a DOM tree, with each tag having its own node. Correctly emulating the DOM tree is important for accurate analysis because many scripts will manipulate the tree to generate links. For example, it is common for scripts to create new "Image" nodes and directly set their URLs. Advertisers also tend to use complex Javascript code to place ads on pages, often going through multiple levels of DOM node creation to load additional scripts. This presumably makes it harder for hackers to replace the advertisements, and for website owners to commit click fraud.

المنارة للاستشارات
www.manaraa.com

To obtain an accurate DOM tree representation, our analysis engine parses each HTML element and creates a corresponding DOM node. This DOM tree is available during script execution. We modeled the interface of our DOM tree after Mozilla Firefox [Mozilla09a]. Updating the DOM tree to also reflect the quirks of other browser DOM implementations is future work. Because we only care about data in HTTP requests and not actually rendering the web page, our DOM tree does not fully implement style and layout interfaces. Ignoring these interfaces makes our DOM implementation simpler and more efficient. The DOM tree also contains hooks for calls that cause the browser to add links to a page. When a script makes such a call, the engine adds the new link URL to either the mandatory or voluntary link library, depending on the parameters. The engine can then filter subsequent HTTP requests that match the dynamically created link URL.

Another option for achieving correct DOM interactions would have been to render HTML and Javascript in a real web browser. We chose not to do this for a few reasons. The first is efficiency. Analyzing every page in a real web browser would require setting up a dummy server to interact with the browser through the local network stack. The browser would also render the entire page and make requests to the dummy server. This adds a significant amount of unnecessary overhead. Our analysis engine cuts out this overhead by directly parsing pages and only emulating parts of the DOM tree that are relevant to leak measurement. A custom DOM tree implementation also makes instrumenting and manipulating of the Javascript interpreter much easier. For example, tweaking the system time or browser version presented to Javascript would require non-trivial patches to a real browser.

5.5.3 Plug-ins and Other Dynamic Content

Javascript is not the only language that enables rich web interaction and can dynamically generate HTTP requests. Popular browser plug-ins like Java [Sun09] and Flash [Adobe09] also have such capabilities. In fact, Java Applets and Flash objects are even more powerful than Javascript. Taking things a step further, stand-alone executable programs may make HTTP requests as well. These applications are free to interact with the user, the local system, and the network in any way that they please.

Correctly extracting all possible links from plug-in objects and executables is undecidable in the general case. This work does not try to analyze plug-ins or dynamic content other than Javascript. In the future, we hope to make some gains by executing plug-in objects in a controlled environment and monitoring their output. It may also be possible to achieve some

المنارة للاستشارات
www.manaraa.com

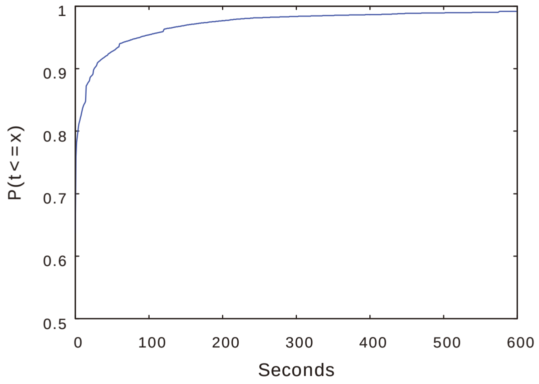


Figure 5.3 Cumulative distribution of delay times for all observed HTTP requests.
P(t<=3) = .794, P(t<=192) = .976, P(t<=3600) = .9996.

improvement through deep inspection and understanding of plug-in objects, but doing so yields diminishing returns because of their complexity and diversity.

Instead of examining dynamic content for plug-in objects, we look at previous requests to create a library of expected URLs. The leak measurement engine compares new HTTP requests that do not match a browser link to the set of *all* prior requests. The closest link is determined by computing the shortest edit distance from a few candidate requests that have the longest matching substring at the beginning. This approach is an effective approximation for finding the closest URL because similar URL strings are much more likely to have common elements at the beginning. The resulting information content is equal to log(m), where m is the size of the library of prior requests, plus the edit distance with respect to the similar prior request, plus two bits to indicate that the request is compared to the library of prior requests and did not come from a link on a web page. In practice, many custom web requests are similar to previous requests. For example, RSS readers and software update services repeatedly send identical requests to check for new data. We can effectively filter most of these messages when measuring information leaks.

5.6 Request Timing Information

In addition to data in the request, HTTP messages also contain timing information. The moment at which a request occurs could be manipulated by a clever adversary to leak information. It is important to consider the bandwidth of timing channels when measuring information leaks. This is especially true for the precise unconstrained measurement techniques in this chapter because they may yield sizes of only a few bits per request in some cases.

The amount of timing information in a request stream is equal to the number of bits needed to recreate the request times as seen by the recipient, within a margin of error. This

المنارة للاستشارات
www.manaraa.com

Scenario	# Reqs	Raw bytes	Simple bytes/%	Gzip bytes/%	Precise bytes/%	Avg. Req. Size
Sports News	911	1,188,317	199,857 / 16.8%	116,650 / 9.82%	13,258 / 1.12%	14.5 bytes
Social Net.	1,175	1,404,268	92,287 / 6.57%	97,806 / 6.96%	12,805 / 0.91%	10.9 bytes
Shopping	1,530	914,420	158,076 / 17.3%	85,461 / 9.35%	6,157 / 0.67%	4.0 bytes
News	547	502,638	74,927 / 14.9%	51,406 / 10.2%	3,279 / 0.65%	6.0 bytes
Web Mail	508	620,065	224,663 / 36.2%	97,965 / 15.8%	3,964 / 0.64%	7.8 bytes
Blog	136	81,162	10,182 / 12.5%	5,534 / 6.82%	262 / 0.32%	1.9 bytes

Table 5.1 Bandwidth measurement results for six web browsing scenarios using four different measurement techniques, along with the average bytes/request for the precise technique.

margin of error is known as the timing interval. It is a short length of time during which the presence of a request indicates a ‘1’ bit, and the absence of a request indicates ‘0’. Using a shorter interval increases the capacity of a timing channel, but also increases the error rate. Previous research on IP covert timing channels found 0.06 seconds to be an effective value for the timing interval in one case [Cabuk04]. This equates to about 16.6 intervals per second.

Prior work on network timing channels looks at IP packets [Cabuk04]. Cabuk et al. describe a channel where IP packets are sent during timing intervals to indicate ‘1’ bits. HTTP requests differ from IP packets in that they tend not to occur as closely together. Instead of having a regular stream of messages throughout a connection, web requests occur in short bursts during page loads, and then at long intervals in between pages. For normal HTTP traffic, we have a *sparse* timing channel in which a vast majority of the intervals are empty.

For a sparse channel, the timing information in each HTTP request is equal to the bits needed to indicate how many empty intervals have occurred since the last request. The cumulative distribution of inter-request delays for our experiments can be seen in Figure 5.3. This shows that that 80% of HTTP requests occur within three seconds of each other, while 95% of requests occur within a minute and a half. Using a variable-length encoding scheme with the first 2–6 bits indicating the length, we can count the timing information in each request as follows (assuming 16 intervals per second):

- Last request <= 3 seconds: **6 value bits**
- Last request <= ~100 seconds: **11 value bits + 2 length bits**
- Last request <= ~50 Minutes: **16 value bits + 4 length bits**
- Last request in past 5 years: **32 value bits + 6 length bits**

This encoding provides a reasonable approximation of the information content in the timing of each request. It is important to note that these figures depend on the number of timing intervals per second. If an attacker can view messages close to the source network, then there may be more than sixteen intervals per second. On the other hand, if a web proxy is configured to

المنارة للاستشارات
www.manaraa.com

increase request jitter, then the number of viable time intervals per second may be less than sixteen.

In this chapter, we assume that HTTP requests are going through a layer-7 proxy or gateway for our timing channel measurements. This means that the only meaningful time is at the start of the request. The timing of subsequent IP packets is controlled by the proxy, not the client, under normal conditions. We believe the presence of a proxy is a reasonable assumption for timing channel measurements. Organizations that care enough about leaks to measure covert timing channels should already have a web proxy in place to mediate outbound information flow (e.g., with data-loss prevention systems [RSA07, Vontu09]).

5.7 Evaluation

We applied the leak measurement techniques described in this paper on web traffic from a controlled environment, and on real web browsing data. The controlled tests involved six 30-minute browsing sessions at different types of websites using a single browser. The real web traffic was collected from ten different people using a variety of browsers and operating systems over a 30-day period. Only data from the controlled scenarios was used for developing the leak measurement engine. None of the live traffic results were used to modify or improve our analysis techniques. We compared the results of our precise unconstrained analysis to incremental gzip compression, simple request analysis, and raw byte counts. The gzip tests involved measuring the amount of new compressed data for each request when using a gzip compression stream that has seen all prior requests and responses. The simple analysis is a technique described in prior research [Borders04] that is stateless and just throws out expected request headers. This section presents our evaluation results, discusses limitations of our approach, and briefly summarizes performance results.

5.7.1 Controlled Tests

We first evaluated our leak quantification techniques on browsing traffic from controlled scenarios. The scenarios were 30-minute browsing sessions that included web mail (Yahoo), social networking (Facebook), news (New York Times), sports (ESPN), shopping (Amazon), and a personal blog website. The results are shown in Table 5.1. The precise unconstrained leak measurements for all of the scenarios were much smaller than the raw byte counts, ranging from 0.32–1.12% of the original size.

المنارة للاستشارات
www.manaraa.com

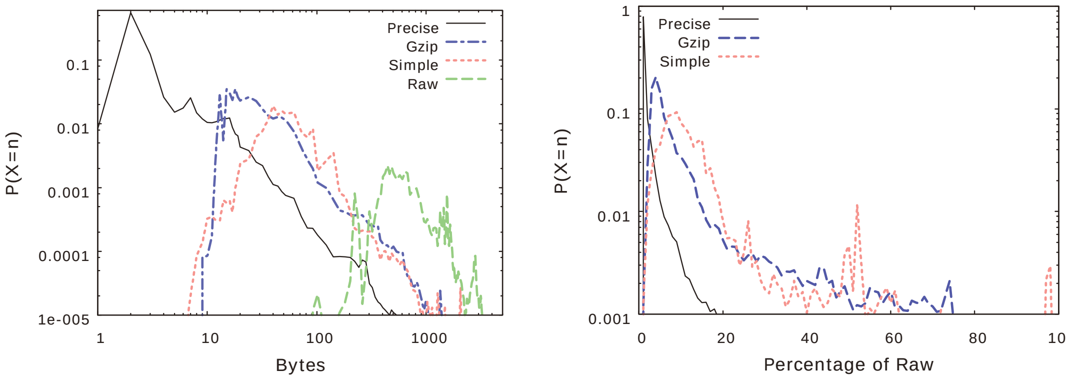
The results were best for the blog scenario because the blog website contained only one dynamic link. The analysis engine was able find an exact match for all of the other requests. Of the 262 bytes that were present in the blog scenario, 118 (45%) of them were from timing information, 86 (33%) from link selection, 48 (18%) from text entered by the user, and 10 (4%) from a Javascript link that contained a random number to prevent caching. The blog scenario represents a near ideal situation for our measurement techniques because we were able to find an exact URL match for all but one request. The resulting average of a few bytes per request serves as a lower bound for standard HTTP traffic. This traffic must at least leak timing and link selection information.

The shopping, news, and web mail scenarios all showed similar precise measurement results. Each of these websites contained a large number of dynamically constructed links that were processed correctly. However, dynamic links often contain information from the client computer. Examples include the precise system time at execution, browser window dimensions, and random numbers to prevent caching. This information must be counted because it cannot be determined by looking at previous requests and responses. From a hacker's point of view, these fields would be a good place to hide data. Opaque client-side state information was particularly prevalent in links for advertisements and tracking images on the shopping, news, and web mail sites.

Precise unconstrained bandwidth measurements for the social networking and sports news scenarios were the highest. The social networking website (Facebook.com) relied heavily on Active Javascript and XML (AJAX) requests that constructed link URLs in response to user input. Because the analysis engine did not trigger event handlers, it was unable to extract these links. The sports news website (ESPN.com) contained a number of Flash objects that dynamically fetched other resources from the web. The analysis engine could not discount these links because it did not process the plug-in objects. In the future, the engine could improve analysis accuracy by obtaining and replaying hints about input events that trigger AJAX requests and dynamic link URLs from agents running the clients. These agents need not be trusted, because incorrect hints would only increase the unconstrained bandwidth measurement.

Gzip compression [Gailly08] was more effective than simple request analysis for all but one of the controlled test cases, but fell far short of the compression level achieved by precise analysis. By running previous requests and responses through the compression stream, gzip was able to discount 84-93% of raw data. URLs and HTTP headers are filled with strings that appear elsewhere in previous requests or responses, giving gzip plenty of opportunities for compression. One benefit that gzip actually has over precise analysis, which was not enough to make a big

المنارة للاستشارات
www.manaraa.com



(a) **(b)**

Figure 5.4 (a) The distribution of precise, gzip, simple, and raw request byte counts for real web traffic. (b) Distribution of request byte counts as percentage of raw for precise, gzip, and simple algorithms.

difference, is that it compresses UI-layer data. Our analysis engine will count the full size of a blog comment, for example, while gzip will compress the comment. Running unconstrained bytes through an additional compression algorithm on the back end may help to further improve precise unconstrained bandwidth measurements in the future.

We did not test generic compression algorithms other than gzip, but would expect only somewhat better results (not an order-of-magnitude improvement). Without protocol-specific processing, compression algorithms are limited in how effective they can be at discounting constrained information.

5.7.2 Quantifying Information in Real Web Traffic

We collected web traffic from 10 users over the course of a month to evaluate our leak measurement techniques. Unlike the controlled scenarios, this traffic came from a variety of web browsers, including Firefox, Internet Explorer, Safari, and Chrome. The traffic consisted of normal daily activity from the volunteers, who consisted of co-workers, friends, and family. The data included 507,505 requests to 7052 unique hosts totaling 475 MB. We also recorded 2.58 GB of response data, not including images, videos, or other binary objects. The web mail request bodies were also ignored to protect privacy. To the best of our knowledge, the collected web traffic did not contain any information leaks from spyware or unusually large uploads that would have negatively skewed the results.

We ran the leak measurement algorithms on the real web traffic one user at a time (the results do not exploit request similarities between users). We first computed the distribution of measured sizes across all requests. Figure 5.4a shows the probability density function of request sizes for raw, simple, gzip, and precise measurements. The precise unconstrained bandwidth

المنارة للاستشارات
www.manaraa.com

measurement algorithm dramatically outperformed the others on real web traffic. The mean precise request size was 15.8 bytes, compared to 132 for gzip, 243 for simple, and 980 for raw. Despite a low average measurement, the precise request size distribution exhibited a heavy tail (standard deviation of 287 bytes). Requests with exact URL matches were usually a few bytes each, while many requests without exact URL matches were a few hundred bytes.

We also calculated the percent reduction in request size with respect to raw measurements. These results can be seen in Figure 5.4b. Again, the reduction is much better for the precise algorithm. Its measurements averaged 1.48% of the corresponding raw values, while the gzip and simple algorithms averaged 9.87% and 13.5%, respectively. The request measurements for the precise algorithm also have a lower variance, with almost all under 20% of corresponding raw values. The simple and gzip size reductions are much more spread out, with some requests measuring 20-75% of the raw size. These requests did not benefit much from gzip or simple analysis.

The unconstrained bandwidth measurement results from real traffic yielded larger values than those from the controlled test cases. The largest average request size of 14.5 bytes from the sports news test was less than the overall average of 15.8 bytes per request for real web traffic. One reason for this is that the controlled tests were not necessarily representative of real web browsing. Other sites that were not in the controlled study may not have exhibited the same mix of requests from plug-ins or event handlers. We did not compute the prevalence of this source of inaccuracy, because doing so would have required manually analyzing a significant portion of the half million requests.

During real web traffic processing, we witnessed a few sources of inaccuracy that were not present in the controlled test cases. One such issue is missing cache objects. Clients may cache resources from the server for long periods of time, making those resources unavailable in a network trace. This is especially problematic for missing scripts that perform important tasks. The effects of this problem could be reduced by having the analysis engine fetch missing objects from the web. However, those objects may no longer be available or might have changed since the original request.

Another source of error only found in real web traffic is the effect of different browser versions. The controlled tests were all performed with Mozilla Firefox [Mozilla09a]. The analysis engine's Javascript and DOM implementation also mirrored Firefox. Real web traffic from other browsers is likely to have different dynamic links corresponding to different browser behavior. These differences could be reduced by implementing other DOM interfaces to match the browser version reported in the headers of each request.

المنارة للاستشارات
www.manaraa.com

5.7.3 Analysis Performance

The real web traffic was analyzed on a commodity laptop computer with a dual-core Intel T2500 processor and 2 GB of RAM. The analysis algorithms ran in a single thread on one core, with the other core being utilized by the operating system. The analysis engine was able process the combined request and response stream at 1.7 Mbps. The bottleneck during processing was CPU. The real web traffic consisted of 3.04 GB of total data, 15% of which (475 MB) was request data and 85% of which (2.58 GB) was response data. The engine processed the requests at an average rate of 0.25 Mbps, and the responses at an average rate of 10.9 Mbps. This disparity in performance is due to the time required to compute the edit distance for request URLs. Javascript execution was included under the response processing time. None of the scripts were given a time limit, and none of them entered infinite loops.

Analysis performance for the prototype implementation would need improvement for use in an intrusion detection system that inspects large volumes of network traffic. One area for optimization is reducing the number of edit distance comparisons and approximating the edit distance computation by only considering multi-byte chunks. Another way to improve performance would be to employ a string co-processor specially designed for edit distance computations. Exploring CPU performance optimizations and maximizing the throughput of the unconstrained bandwidth measurement engine is future work.

The memory footprint during analysis was quite large for the prototype implementation. It kept all of the observed links in memory and did not attempt to free older data that was less likely to improve analysis results. Processing 20 MB of web browsing traffic from one user during a single day required 120 MB of RAM. Although this would be unreasonably large for an intrusion detection application, we believe that this number could be greatly reduced by simply discarding links from old pages. While analysis results may be a little bit worse, the number of links that are loaded from pages that have been open for hours is far smaller than links that are loaded from recent pages. Another possible optimization is sharing link information across users.

5.8 Entropy Mitigation Strategies

The evaluation showed that a significant portion of information in web requests must be counted because it originates from entropy on the client. If this entropy can be reduced or measured at a trusted source, then the analysis engine can obtain more accurate results. This section discusses possible strategies for reducing inaccuracies in unconstrained bandwidth measurements due to entropy on client computers.

المنارة للاستشارات
www.manaraa.com

5.8.1 System Information and Human Input

The current leak measurement engine cannot see actual system information or human input to a client; it only witnesses the resulting requests. Due to the complexity of active content on websites, system information and human input can sometimes lead to a chain of events that generates a much larger output than the size of the original information. For example, clicking on a particular place on a web page may lead to an AJAX request that contains a few hundred bytes of XML. Speculatively firing events would help somewhat with determining expected requests, but such an approach would quickly lead to an exponential blow-up in processing time. A better solution would be to obtain system information (screen resolution, OS, installed plug-ins, etc.) and human input hints from an agent running on the end host. This agent could be a browser plug-in that records and sends all of the system information and human input events to the analysis engine. Instead of having to speculate, the engine could then replay the exact sequence of inputs and verify that the output is the same. It could only count the size of the original input, rather than the larger resulting output. It is also okay if the agent reports data incorrectly, because doing so would only increase the unconstrained bandwidth measurement and raise suspicion.

Depending on the threat model, it may also be possible to reduce unconstrained bandwidth measurements by discounting human input entirely. This approach may be appropriate if the user is trustworthy, but malware is a concern. A trusted device, similar to a hardware key-logger, could intercept mouse and keyboard events before they reach the computer, and then report them to the leak measurement engine. This would aid analysis in a similar manner as a hint from a browser plug-in, except that the size of the original human input could be discounted as well, assuming that the user is trusted.

5.8.2 Timing

The timing of each request has the potential to leak several bits of information to an observer stationed outside of the network. The traditional method for mitigating timing channels is to add entropy to each request. For web traffic, this can be achieved by adding a trusted proxy server between the client and the web server. This proxy can add jitter to each web request by delaying it a random amount of time. This could significantly increase the size of the timing interval, raising it from 0.06 seconds to 1 second (any more might disrupt usage). Randomly delaying requests up to 1 second would reduce the amount of timing information in each request by 5 bits, which can add up to a significant savings for a large number of requests.

Another option available to us that would not be feasible for mitigating a traditional IP packet timing channel is reducing the total number of requests. Every time a client makes a

المنارة للاستشارات
www.manaraa.com

request for a web page, a smart caching proxy could pre-fetch all of the mandatory links. Then, when the client requests a resource from a mandatory link, the proxy can return the result without any information leaving the network, thus precluding leakage through those requests.

In addition to the timing of requests themselves, some requests include an explicit time value. This is the system time at which a script executed on the end host. Websites may include this time value to prevent caching, or to collect statistics about latency from their users. In any case, it differs slightly from the time that a request actually appears on the network, has a high precision, and can therefore leak information. A proxy server can eliminate timing information of this form by discovering it with the edit distance algorithm and then overwriting it with the time that the proxy actually sends the request.

5.8.3 Random Number Generator

Many websites have scripts that include random numbers in link URLs. The purpose of doing this is to prevent caching. At the same time, however, these requests leak data in their selection of random numbers. One way of reducing entropy from the random number generator (RNG) is to instead have a network service that handles random number generation. When an executing script makes a call to fetch a random number, the Javascript engine could request a new random number from a trusted central location instead of using the local RNG. This would move random numbers from the set ***U*** of UI-layer input to the set ***I*** of network inputs, allowing the analysis engine to discount them from the information measurement in outbound web requests (assuming they are not modified by malware).

5.9 Conclusion and Future Work

This chapter introduced a new approach for quantifying information leaks in web traffic. Instead of inspecting a message's data, the goal was to quantify its information content. The algorithms in this chapter achieve precise results by discounting fields that are repeated or constrained by the protocol. This work focuses on web traffic, but similar principles can apply to other protocols. Our analysis engine processes static fields in HTTP, HTML, and Javascript to create a distribution of expected request content. It also executes dynamic scripts in an emulated browser environment to obtain complex request values.

We evaluated our analysis techniques on controlled test cases and on real web traffic from 10 users over a 30-day period. For the controlled tests, the measurement techniques yielded byte counts that ranged from 0.32%-1.12% of the raw message size. These tests highlighted some

المنارة للاستشارات
www.manaraa.com

limitations of our approach, such as being unable to filter parts of URLs that contain random numbers to prevent caching. For the real web traffic evaluation, the precise unconstrained byte counts averaged 1.48% of the corresponding raw values. This was significantly better than a generic compression algorithm, which averaged 9.87% of the raw size for each request.

In the future, we plan to implement similar leak measurement techniques for other protocols. E-mail (SMTP) will probably be the most challenging because a majority of its data is free-form information from the user. There is also a lot of room to improve the dynamic content analysis techniques. Obtaining user input hints from clients and executing plug-in objects can help extract additional request URLs. Finally, we hope to optimize and integrate the techniques from this paper into a network intrusion detection system that uses bandwidth thresholds to discover information leaks.

المنارة للاستشارات
www.manaraa.com

CHAPTER 6

INFERRING MALICIOUS ACTIVITY WITH A WHITELIST

6.1 Overview

Traditional threat detection approaches involve directly categorizing and identifying malicious activity. Examples of this methodology include anti-virus (AV) software, intrusion detection systems (IDSs), and data loss prevention (DLP) systems. These systems rely on *blacklists* that specify undesirable programs and network traffic. Blacklists have a number of benefits. First, when some malicious activity matches a signature on a blacklist, an administrator immediately knows the nature of the threat and can take action. Second, many blacklists are globally applicable and require little tuning for their target environment (a known computer virus is unwanted in any network). Widespread applicability also goes hand in hand with low false-positive rates; activity that matches a blacklist is usually not of a legitimate nature. These advantages, along with the simplicity and speed of signature matching, have made blacklisting the most prevalent method for threat detection.

Despite its benefits, blacklisting suffers from fundamental limitations that prevent it from operating effectively in today's threat environment. One limitation is that a blacklist must include profiles for *all* unwanted activity. Malicious software (malware) is now so diverse that maintaining profiles of all malware is an insurmountable task. Research shows that even the best AV software can only detect 87% of the latest threats [Oberheide07]. Furthermore, a hacker who targets a particular network can modify his or her attack pattern, test it against the latest IDS and AV signatures, and completely avoid detection, as is demonstrated in [Vigna04].

In this thesis, we explore an alternative approach of detecting malicious network activity using a combination of application identification techniques, described in Chapter 4, and a whitelist. The goal of our whitelist is to classify and categorize all legitimate network activity. Anything that does *not* match the whitelist is considered suspicious. This approach eliminates the

www.manaraa.com

need for scaling signature generation efforts with respect to attack diversity. Instead one only has to keep track of legitimate application behavior, which is easier because good applications do not try to hide by frequently changing their profiles. Whitelisting is further advantageous because is it able to identify new and unknown threats. *Any* network traffic that does not fit the profile of a legitimate application will generate an alarm.

Even though network-based whitelisting is a promising method for threat detection, there are significant challenges that must be overcome before it is practical in a production environment. First and foremost is building an effective whitelist. If there are gaps in the whitelist, then false positives will cripple the detection system. Whitelists also evolve over time as users install and upgrade their applications. So, adding new entries to the whitelist must be straightforward for security analysts and not require assistance from an engineer.

Another major challenge of whitelisting is specifying legitimate activity with enough detail that an attacker cannot trivially *mimic* good behavior. Mimicry attacks are impossible to prevent in the general case, but an effective whitelist will make it difficult to mimic good behavior while still conducting nefarious activity. Whitelists should also put a hard limit on the damage that can be done by malware, forcing it to "behave well" in order to avoid detection. An example of a poor whitelist would be one that allows all outbound network traffic on specific TCP ports. The amount of work required for malicious software to avoid detection by communicating over an allowed TCP port is next to nothing, and the amount of data it can send over that port is unlimited. An optimal whitelist will only contain the minimum set of activity needed for a given application to function properly.

In this chapter, we present a whitelisting approach that is based on methods from the previous chapters for detecting web applications and measuring their outbound bandwidth. Chapter 4 talks about how to identify network traffic that was generated by automated web applications rather than by humans browsing the web. Application of these methods results in a list of *alerts* that specify the way in which particular traffic differs from human web browsing (timing, formatting, etc.) along with the traffic content and host name. Chapter 5 discusses a method for quantifying outbound information flow in web traffic. This method, when combined with graduated bandwidth thresholds, also generates alerts in response to abnormally large information flows.

Our whitelist consists of a mapping from alerts to known applications. All of the entries in the whitelist for a particular application make up its *application profile*. Each whitelist entry may match a number of alerts based on the server name, server address, client address, type of

المنارة للاستشارات
www.manaraa.com

alert (bandwidth, timing, formatting, etc.), message field value (e.g. user-agent or header field for non-browser requests), or amount of bandwidth usage.

We created the whitelist with application profiles based on observation of network traffic from over 700 computers in a corporate network. At the end of a two-year deployment, the whitelist contained 501 different application profiles with a total of 1789 entries. As the whitelist matured towards the end of the deployment, the number of new whitelist entries per month approached about 50, or 1.6 per day on average. These entries were the result of about 10 false positives per day, which is a reasonable rate given the size of the deployment. We expect that a more widespread deployment will have an even lower false rate of whitelist entries per computer, as there will be more overlap for common applications. The test deployment also helped us understand more about the network behavior of different legitimate and malicious programs.

A key difference between whitelists and blacklists is that the content of a whitelist may vary significantly from one deployment to the next. Some organizations with strict policies may only allow a specific subset of web applications on their network. Others may have more relaxed policies, but still not want particular applications, such as file sharing or instant messaging programs, running on their network. When an organization initially deploys a whitelist-based threat detection system, they can tune the whitelist by removing application profiles for unwanted programs. These profiles will still remain in the system to aid in threat remediation, but alerts that match these entries will be displayed to an administrator instead of being ignored by the system.

6.2 Prior Whitelisting Systems

The concept of whitelisting is not new to the field of computer security. Prior research on intrusion detection using sequences of system calls [Hofmeyr98] looks at trusting known good behavior at the system call API layer to isolate malicious execution patterns. The authors were able to reliably detect a number of intrusions that led to sequences of system calls not seen during normal activity, while maintaining a low false positive rate. The seminal work by Hofmeyr et al. led to a figurative arms race between researchers finding new ways of exploiting a system while mimicking legitimate system call behavior [Garfinkel03b, Wagner01], and those developing more precise characterization methods that make mimicry more difficult, such as examining the entire stack trace for each call [Feng03]. An important result of this escalation between attack and defense technology is that mimicry attacks are extremely difficult to prevent altogether, but one can make them much more difficult with a precise whitelist-based detection system.

المنارة للاستشارات
www.manaraa.com

The research presented in this chapter is similar to system call-based intrusion detection in that they both use whitelists to identify threats. However, network traffic is a different input domain. Whitelists for system call IDSs precisely enumerate the set of all allowed call patterns. Taking this same approach and explicitly specifying the set of all benign network traffic would be a very difficult task due to the huge diversity of possible messages. Instead, we are taking meta-information in the form of alerts about traffic that deviates from a conservative baseline, and then trying to determine their source application. The methods we use for alert generation also take session state into account, which is critical because it directly influences the set of expected network messages at any given time. System-call IDSs only consider a very small amount of session state in the form of call sequences. Restricting examination of network traffic to only the last several messages in this manner would preclude any sort of bandwidth or regularity measurement. Successful application of a whitelisting approach to network traffic requires complex long-term state tracking at the front end to generate meaningful statistics from which the whitelist entries can classify traffic according to its source application.

A classic example of whitelisting for security purposes is a firewall with specific *allow* rules followed by a *deny all* rule. In this scenario, only traffic associated with known legitimate ports and protocols is allowed to pass in or out of the network. Although this type of whitelist is very effective at protecting internal services from probing and attack, it is not effective at blocking malicious applications from accessing the Internet. This is because it is trivially easy for malware to use an outbound port and protocol allowed by the firewall, and then enjoy a virtually unlimited communication channel to the external network.

Some security systems go a step further than firewalls and actually determine the application-layer protocol for each network connection [Netwitness09, Sandvine09]. This way, they can identify programs that are trying to communicate over a standard TCP port for one application-layer protocol, such as 80 for HTTP, using a different application-layer protocol, such as SSL (secure sockets layer), SSH (secure command shell), or IRC (internet relay chat). This type of whitelisting approach does help detect some unwanted activity. However, it fails to meet the requirements of an effective whitelisting system in that it is trivially easy for malware to communicate over an allowed port using an allowed protocol. Furthermore, forcing the use of an allowed protocol puts no limit on the amount or outbound traffic or its content, provided that it loosely conforms to protocol specifications.

المنارة للاستشارات
www.manaraa.com

Timestamp	Client Addr.	Server Addr.	Server Name	Alert Type	Alert Details
5:12 PM	10.0.0.100	10.0.29.64	www.website.com	Header	X-my-header
11:09 AM	10.0.0.100	10.0.1.200	server.othersite.net	Regularity	c.o.v.: 2.718
10:22 AM	10.0.0.102	10.0.63.69	www.mysite.com	User-Agent	MyHTTPAgent
4:15 AM	10.0.0.105	10.0.14.71	---	Bandwidth	1,618,034 bytes

Table 6.1. Four sample alerts that indicate various formatting, timing, and bandwidth anomalies. The server name may not be present for hosts without DNS entries.

6.3 Whitelist Design

The purpose of a whitelist is to provide a mapping from alerts to known applications. The whitelist consists of entries, each of which has two parts. The first is the matching section where the whitelist entry specifies which alerts it will match. The matching section does *not* reference raw HTTP requests that caused the alert because they may not always be available in practice due to privacy or performance considerations. Allowing whitelist entries to reference request URLs and content would also increase complexity and may make it more difficult for an average security analyst to update the whitelist. The second part of a whitelist entry is the action, which can associate alerts that match the entry with a particular application or ignore them entirely. The set of all whitelist entries that associate alerts with a particular application makes up an *application profile*.

The alerts that we consider in whitelist entries come from the formatting, timing, and bandwidth analysis techniques presented in earlier chapters. The contents of an alert can be seen in Table 6.1. Each alert contains fields specifying the time, client address, server address, server name, alert type, and alert details. The server name may be taken from the "Hostname" header field in an HTTP request. The alert type is one of a fixed set of values denoting the type of anomaly. Alert types include: *unknown user-agent*, *unknown header field*, *bad header format*, *regularity*, *delay time*, *time of day*, and *bandwidth*. There are also alert types for sub-classes that indicate different measurement methods and thresholds, including *8-hour c.o.v regularity*, *unknown Mozilla/4.0 user-agent field*, *bandwidth level 2*, etc. Finally, the details may contain an arbitrary string describing the exact nature of the alert. For formatting alerts that indicate an unrecognized field, the details hold its value. For timing and bandwidth alerts, it details show the exact regularity, delay, or byte count measurement.

The module that generates alerts is responsible for verifying the server name with a reverse DNS look-up; we assume it is correct at the whitelisting stage. The alert generation module also does not currently take any measures to prevent DNS hijacking. One possible counter-measure to DNS hijacking would be to raise an alarm if the IP address for a hostname in

المنارة للاستشارات
www.manaraa.com

Time	Client	Server	Server Name	Type	Details	Application
*	*	*	sb.google.com	All Timing	*	Google Toolbar
*	*	*	*	User-Agent	GoogleToolbar \d+\.\d+\.\d+	Google Toolbar
*	*	*	sqm.microsoft.com	Bandwidth-1	*	MS Office
*	*	9.73.8.x	*	All Timing	*	Yahoo Msngr
2-3 AM	IDS	*	*.snort.org	User-Agent	wget	—

Table 6.2. Five sample whitelist entries specifying legitimate network behavior. The first six columns dictate alerts that the entry will match, and the **Application** specifies association of those alerts with an application, or ignoring them altogether (“—”). Entries with “All Timing” match alerts from traffic with regular timer-driven requests to the given servers. “User-Agent” entries match HTTP requests that have a given user agent (regular expressions are allowed). The “Bandwidth-1” entry matches bandwidth alerts that exceed the first bandwidth threshold.

the whitelist changes to an IP address in another sub-network owned by a different autonomous system (AS).

Entries in a whitelist may reference any of the alert fields. For each field, a whitelist entry may match an exact value, all values, or a subset of values. Table 6.2 contains examples of several whitelist entries. The timestamp field can have a single absolute time range, or it can contain a daily time range along with a mask specifying certain days of the week. This is helpful for whitelisting automated processes, such as updates (the fifth entry in Table 6.2 allows IDS signature updates), that run on a fixed schedule. The client address and server address fields support address ranges. These are helpful for specifying clients with different security requirements or services that run on a sub-network of IP addresses. The server name field may contain a hostname wildcard matching string (e.g., “*.website.com” will match alerts for any domain name ending in website.com). There are only a small number of alert types corresponding to different kinds of formatting, timing, and bandwidth anomalies, so the whitelist entries may have a bit mask matching any combination of alert types. Finally, a whitelist entry may use a full-fledged regular expression to match the alert details field. Regular expressions are particularly helpful when matching new message fields that include a frequently changing application version string, as can be seen with the “GoogleToolbar \d+\.\d+\.\d+” detail matching string in the second entry of Table 6.2.

6.4 Whitelist Construction Methodology

Now that we have a method of specifying whitelist entries, it is essential that we outline a systematic approach for generating new entries that is straightforward and comprehensible to an average security analyst for most cases. The process begins when there are new alerts that do not match any current whitelist entries. The alerts can fall into one of three categories – formatting, timing, or bandwidth – which influences the approach that should be taken when creating a new

المنارة للاستشارات
www.manaraa.com

whitelist entry. Often times, one application will generate alerts from multiple categories, such as a timing alert and an unrecognized header field alert. This section describes methodology for grouping alerts, determining their source application, and creating appropriate general-purpose whitelist entries. It also discusses creating domain-specific whitelist entries and security considerations associated with whitelist construction (i.e. how to make sure whitelist entries do not open up a backdoor that allows hackers to circumvent the system).

6.4.1 Grouping Alerts

The first task in constructing whitelist entries is determining which alerts are associated with the same application. It is best to group alerts first by server domain, and then by time and client to figure out which ones are associated with the same application. Most of the time, alerts for the same application will all have the same server domain name. However, this is not always the case for domains that host several applications (e.g., google.com, microsoft.com, yahoo.com, etc.) or for secondary application servers that only have an IP address (e.g., instant messaging servers, peer-to-peer applications, etc.). In these situations, clustering alerts by client and by time provides a strong indication that they are associated with the same application.

6.4.2 Determining the Source Application

After the set of alerts associated with one application has been determined, the next step is identifying that application. One must exercise caution during this part of the process, so as not to add a whitelist entry for malicious network traffic. During our experience constructing a whitelist, we encountered several cases where an alert looked benign (primarily for formatting alerts) but contained subtle differences or omissions compared to normal traffic. An example is the user-agent field "compatible" that is present in any standard browser HTTP request. A particular spyware program mistakenly included the field as "**C**ompatible", which only differs in the capitalization of the first letter.

In some cases, alerts may be associated with a general class of applications or operating systems and not be specific enough to identify a particular application. An example of this would be a header field that contains "Windows" or "Linux" for the details. In these cases, the source application for the whitelist entry should be left empty, indicating that matching alerts are too generic to say anything meaningful about the source application.

The best approach for accurately identifying the source application for a group of alerts is to consider a number of data points and make sure that they are consistent with one another:

المنارة للاستشارات
www.manaraa.com

- **Research the server.** If the host is running a public web server, then visit pages on the site (using a browser sandbox in case it is malicious) to learn more about the server. If not, perform a WHOIS look-up to determine who owns the server and where it is located. In particular, look for any software products hosted on this server that may download updates or report registration/usage information. Also look for Java applets with security certificates that are hosted on the server, as they may send arbitrary data over the network.
- **Research the message formatting.** What do the requests look like? Specifically, focus on the "User-Agent" field for HTTP requests, which is supposed to identify the client application. Clients may sometimes impersonate a web browser, but will usually use a string describing the application. For example, the second whitelist entry in Table 6.2 resulted from alerts with user-agent values containing "Google Toolbar". If the user-agent value contains a unique word but not an easily recognizable application name, then querying a search engine often yields information about the application.
- **Examine the request URL.** The URLs of HTTP requests often give a clue as to their nature. Examples include "/update", "/reporting", "/rssfeed", etc. This can help narrow down the type of application making requests.
- **Examine the message content.** This is especially important for bandwidth alerts. What information is the client sending to and receiving from the server? In most cases, it will be directly apparent what type of data is contained in HTTP messages (images, documents, etc.) just by looking at the content-type header, as they should not contain encrypted information. If encrypted of obfuscated data is present, it usually indicates proprietary usage reporting for outbound requests, or program updates for inbound requests.

Methodically examining these four information sources for each set of alerts associated with a particular application will usually lead to a strong conclusion about the source application and whether it is legitimate. For rare cases where messages have cryptic content and go to an unknown or suspicious server, it is best to perform further forensic analysis on the client to assess the legitimacy of the application generating the alerts.

6.4.3 Creating the Whitelist Entries

Once the source of alerts has been determined, the last step is creating appropriate whitelist entries that will match future alerts from the source application. The goal of these entries is to match *all* alerts that the application can generate, and not match alerts generated by *any* other application. Defining perfect whitelist entries can be difficult, or even impossible in cases where applications partially overlap. Due to the base-rate fallacy [Axelsson00], it is best to err on the

المنارة للاستشارات
www.manaraa.com

side of fewer false positives, unless the likelihood of illegitimate activity generating an alert is comparable to the likelihood of a false positive. For practical purposes, the whitelist entries should match the approximate minimal set of alerts that includes *all* alerts an application can generate. The precise minimal set may be overly complicated, leading to false positives and not offering much extra security. For example, there may be 100 servers in a 255-address contiguous sub-network. It is sufficient to include the entire sub-network in a whitelist entry instead of enumerating every address.

Given a set of alerts for an application, the first step in building a whitelist entry is to determine all of the alert types and alert details that the application may generate in the future. In almost all cases, it is sufficient to include the exact alert types and alert details present in the current set of alerts. The primary exception to this rule is when there are numerous formatting alerts that contain slightly different details, such as different version numbers. In these cases, one should use a regular expression that matches all alerts by inserting placeholders for dynamic values. An example of this can be seen in the detail column of the second alert entry in Table 6.2. The detail string "GoogleToolbar \d+\.\d+\.\d+" matches any message field that starts with "GoogleToolbar" and contains a three-number dot-separated version number.

The final thing to consider before making the whitelist entry is whether alerts are only associated with a particular server or domain postfix. This must always be the case to reason about alert types with non-specific details, such as timing and bandwidth alerts. If the alerts are all associated with a particular server or set of servers, then the whitelist entries should reflect that fact. This is likely to be the case for web applications that only communicate over the network to report usage statistics or download updates. However, many internet applications, such as browser plug-ins and utilities like GNU Wget [Niksic98], may access a wide variety of servers and should not have a specific server field.

6.4.4 Deployment-Specific Whitelist Entries

So far, we have not discussed the timestamp and the client whitelist fields. These fields should always be left blank for general-purpose whitelist entries that are applicable in any environment. However, client and timestamp settings may be appropriate for a specific deployment. The client field is useful if only certain computers or sub-networks of computers are permitted to run a particular application. For example, web developers may have free reign to send messages with arbitrary formatting to company servers, while these messages may be indicative of an attack when coming from different clients. Another example, which also incorporates the timestamp, can be seen in the last line of Table 6.2. This whitelist entry specifies

المنارة للاستشارات
www.manaraa.com

that the server with hostname "IDS" is allowed to access *.snort.org with the Wget application [Niksic98] between 2 AM and 3 AM, presumably to fetch updates. It is important not to allow Wget for other computers or at other times, because while it is a legitimate utility, attackers often call it from shell code to download secondary malware payloads [Borders07]. Deployment-specific whitelist configuration can provide extra security, and should be part of the initial deployment process for networks with heterogeneous client security policies.

6.4.5 Whitelist Security Considerations

In general, servers fall into one of two categories: *small* or *well-known*. A small server is any server that is not backed by a well-known and reputable organization. Although they almost always mean well, small servers are more susceptible to hackers. When creating whitelist entries for alerts that could potentially involve small servers, it is important avoid bandwidth alerts whenever possible and use the minimum threshold. If it is public knowledge that the whitelist for a popular security system allows a huge amount of bandwidth to some boutique website, it may become a target for hackers who wish to circumvent the security system.

Though it is important to be wary of small servers, there are also security considerations for well-known servers. Hackers may post data on a well-known server that they do not control, and retrieve the data remotely. An example is spyware that reports sensitive information back to its owner is by sending it in an e-mail message from a web mail account. Theoretically, a hacker can use any website at all that accepts posts and will display them back at a later time. When creating whitelist entries for well-known servers that allow posting of data, it is important to only match the minimal set of alerts. For example, if HTTP requests to a web mail server with various user-agents are generating alerts, it is best to enumerate the specific user-agents that may access the server rather than trust all user-agents. This increases the likelihood of detecting malware that uses the same web mail service for sneaking data out of a compromised machine. As with alerts to small servers, it is also critical to limit bandwidth whitelist rules for well-known servers that accept posts. This minimizes damage that can be done by an attacker who knows the whitelist rules.

The last thing to keep in mind while generating whitelist entries is that the network path between clients in an enterprise and servers on the Internet is not safe. Regardless of how trustworthy a server may be, whitelists should *never* contain entries that are unnecessarily broad, unless they are deployment-specific and only apply to servers within the local network.

المنارة للاستشارات
www.manaraa.com

6.5 Case Study: Constructing a Whitelist for a Corporate Network

We deployed a network monitoring system that generates alerts based on algorithms outlined in this thesis in a medium-sized corporate network with over 700 computers over a two-year period from February 2007 to February 2009. This deployment was not wide-scale enough to draw conclusions about the precise rate of false positives and new whitelist entries. However, it does demonstrate the feasibility of a whitelisting approach, and leads to a better understanding of both legitimate and malicious program behavior as it relates to web traffic. This section first shows the rate of whitelist entries over time, and then discusses characteristics of different legitimate and malicious programs.

6.5.1 Whitelist Entry Rate

The deployment monitored traffic at the network edge for two years. While traffic was collected steadily throughout this time, there were gaps in alert reporting due to data loss. We analyzed alerts that were collected and manually constructed the resulting whitelist entries according to the principles outlined in the previous section. Figure 6.1 shows the number of new whitelist entries per month over the two-year time period. Instead of measuring the time at which each whitelist entry was added, this graph measures the time of the first alert that matches each whitelist entry. The rate at which new whitelist entries are needed is much higher during the first few months of the deployment, peaking at 350 during the fifth month, and then it steadily declines to about 50 entries per month towards the end of the deployment period. This steady rate of new entries corresponds to the regular appearance of new applications and application versions. We believe that a steady rate of new whitelist entries that hovers around 1.6 per day is reasonable for a network of over 700 computers, and is likely to take an analyst only a few minutes to update every day. For the last two months of the study, this rate of whitelist entries corresponded to about 10 false positives per day for the 700 computers. Although we have not tested whitelisting in a larger network, we expect that the amount of work will scale in a less-than-linear fashion due to increasing overlap in whitelist entries.

The security policies in the test corporate network were almost completely open. Users were allowed to install just about any program they wished on their computers, including instant messaging applications, anti-virus software, and web browsers. The only exceptions were programs used for illegal activity or that take up excessive network bandwidth, such as file-sharing software. Developing an effective whitelist is the most challenging in an open network

المنارة للاستشارات
www.manaraa.com

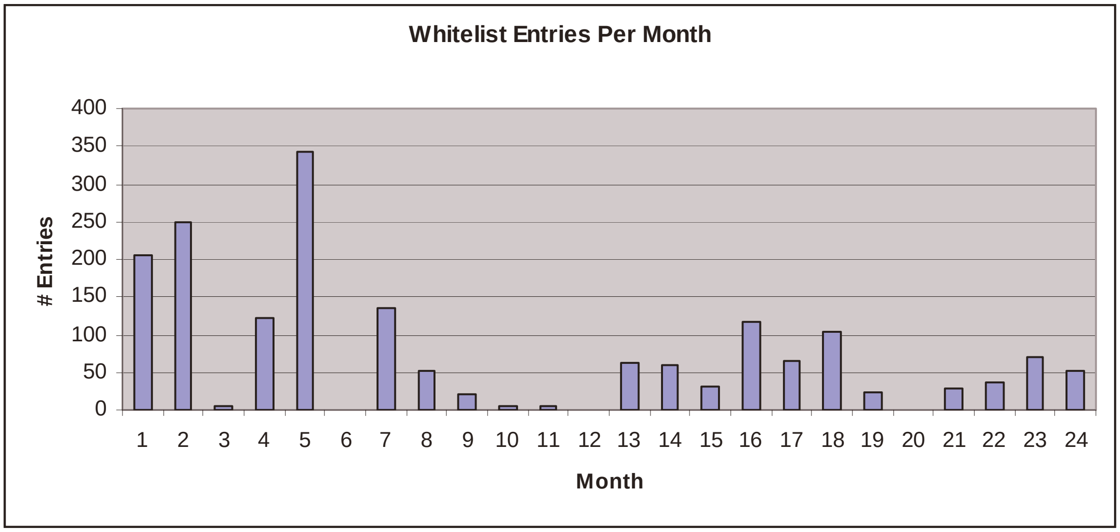


Figure 6.1 New whitelist entries per month over a two-year test deployment. Months with few or no entries correspond to data loss. Toward the end of the deployment, the rate of whitelist entries approaches 50 per month.

environment. We expect that applying our whitelisting methodology in a closed network with a more limited software base would lead to a smaller whitelist and a lower false-positive rate.

The results shown in figure 6.1 contain some anomalies. One would expect the rate of new whitelist entries to always be highest during the first month of installation, and then die down in an exponential fashion. In our test deployment, the number of entries peaked at the fifth month. The reason for this is that the frequency of data loss in the first few months was also higher because the system was in an earlier phase of testing. If we were able to go back and evaluate the whitelist on a complete packet capture for the entire duration of the two-year period, then we would expect to see a peak at the beginning, and a gradual decline in entries throughout.

6.5.2 Whitelist Entry Characterization

The whitelist entries that were added during the test deployment fall into a number of general groups according to the type of traffic they match.

Formatting Only – Most of the whitelist entries fall into the formatting-only category. These entries will match web requests and alerts solely on their formatting. Specifically, these entries match values in the User-Agent field, and custom header fields. The reason that these entries occur so frequently is that most legitimate applications uniquely identify themselves by setting a custom user-agent value, and some use custom HTTP headers. Because these headers and user-agent values are unique, there is no need to specify a particular host name or range of server IPs in the whitelist entries; every request that matches the given format should come from

www.manaraa.com

the associated application, unless it is being intentionally spoofed. Applications in this category also do not generate large or frequent enough requests to trigger bandwidth or timing alerts.

No Formatting – Many applications send out HTTP requests that have no custom header fields or user-agent values. This is just as anomalous as a custom user-agent, but it is not as easy to identify which application generated a request with no identifiable formatting. In this case, we rely on the server name or IP address to dictate the source application. Whitelist entries in this group will specify "No User-Agent," as well as a server name/IP matching string, and the associated application.

As far as we saw, only one type of application would send requests without a user-agent to servers other than those owned by its vendor. These were RSS readers. We created a special case module for RSS readers that peeked at the content of server responses to identify the content as an RSS feed. If a client sends a request without a user-agent to a server that responds with an RSS feed, then that request is classified as coming from an RSS reader.

Timing – Some applications used regular requests to communicate with their home servers. These requests triggered the timing or regularity filters described in Chapter 4. We found that distinguishing between timing and regularity did not offer much benefit to the whitelist entries, because applications that trigger one usually triggered both. A number of applications that triggered timing alerts made requests with generic user-agent and header values, so were identifiable only by timing alerts to their home server. An example of a request with a generic user-agent is when an application makes a call to the standard Windows internet library to fetch a URL. One version of Windows will make the call with the user-agent "WinInetRequest 1.0". Like whitelist entries for requests with no formatting, the timing entries were based solely on the destination server IP or hostname. Many of the timing whitelist entries were associated with weather reporting applications. We did not witness any legitimate software making timer-driven requests with non-descriptive user-agents to websites other than those owned by their vendors.

Timing Plus Formatting – There were a few applications that would trigger timing alerts to a wide variety of server IP addresses on the Internet. The prime example of this was a messaging application that also supported voice chat. Presumably, timing alerts to random IP addresses came from digital voice conversations. In this case, the whitelist entries would match any timing alerts where the request formatting also matched a specific value (that of the messaging software). This way, other applications that generate timing alerts to random IP addresses do not match the whitelist.

Bandwidth – Bandwidth whitelist entries are the most security-sensitive. If entries exist that allow arbitrarily large amounts of outbound traffic, then hackers could exploit such entries to

المنارة للاستشارات

www.manaraa.com

steal data. Only a few entries were added to the whitelist for bandwidth alerts, and they were for widely-deployed applications, such as operating system update utilities, office application usage reporting, desktop search, and voice clients. Bandwidth entries matched both the server name and request formatting. So, for example, only bandwidth alerts to update.microsoft.com that also had the user-agent “Windows Update” were ignored.

6.5.3 True Positives

During the evaluation period, we also encountered a number of true positives that did *not* match entries in the whitelist. It is useful to understand what types of anomalous behavior the whitelist does detect in practice, so that we have a better idea of what to look for when evaluating whether an alert is a false positive.

The simplest type of true positive is one that occurs from formatting only. Some unwanted programs will explicitly identify themselves because they may have a legitimate purpose in the eye of their providers. Adware programs that annoy the user but do not steal personal data fall info this category. File sharing programs are also considered unwanted in many networks. Another more nefarious example is “Remote Administration” software, which is essentially a backdoor that gives an outsider full control of a computer. Presumably, the outsider is an “administrator” trying to fix a problem, but could in fact be a hacker exploiting the target. Some attacks also make use of Wget [Niksic98] to fetch their secondary payloads out of convenience. These attacks are detected immediately.

The next type of software that can be detected by formatting alone is poorly-written malware. There are several examples where a malware writer selects a user-agent or header field that is anomalous by mistake, when they are actually trying to hide. Some omit the user-agent field entirely, which also generates an alert. As mentioned earlier in this chapter, one malicious program was seen with the user-agent field “Compatible” instead of “compatible”. Other such screw-ups include spelling “Referer:” as “Referrer:” and sending a user-agent that reads: “Internet Explorer”, when the real Internet Explorer user-agent is: “Mozilla/4.0 (Windows; MSIE 6.0)”. These blunders are easy to avoid from an attacker’s perspective, but detecting them is an immediate way to identify haphazardly written malware.

Some malware that uses protocols other than HTTP will try to avoid detection by running those protocols over the standard HTTP port 80. This is seen most frequently with internet relay chat (IRC), which has become the preferred language of command and control for many botnets. Any traffic that deviates from the HTTP protocol will immediately generate an alert.

المنارة للاستشارات
www.manaraa.com

Other malicious programs were seen triggering delay time and regularity alerts on several occasions. Examples include malware that frequently called home by connecting to a web mail account, as well as the previously mentioned remote desktop software. These programs were detected by virtue of their continuous interaction over the network. It is somewhat harder for hackers to avoid timing filters, because doing so requires a change in the behavior of their software, not just modifying the format to mirror a legitimate web browser request.

None of the true positives from malicious software that we witnessed during the evaluation period triggered bandwidth alerts. However, alerts were generated whenever a user uploaded large amounts of data. As far as we know, none of these uploads were associated with insider leaks. They mainly consisted of documents and pictures, with the occasional large message board post. In a larger network, manually inspecting all the file uploads to the Internet could get expensive. One way to deal with these alerts is to isolate and extract file uploads and text area posts, then send them to a data-loss prevention (DLP) system for content analysis. Then, analysts would only need to inspect alerts for bandwidth violations that occurred as a result of *application* data, rather than data inserted by the user on a website post.

6.6 Conclusion and Future Work

In this chapter, we examined a new approach to classifying activity from network applications using a whitelist. This chapter provided examples of whitelist entries that match alerts from earlier chapters for specific applications. We also described methodology for effectively creating new whitelist entries without opening up backdoors for malicious software.

We evaluated the whitelisting approach by deploying it in a corporate network with over 700 computers for a two-year period. This case study shows a reasonable rate of new whitelist entries and false positives over time of about 1.6/day and 10/day, respectively. The case study also highlights different categories of network behavior from both legitimate and malicious applications. Overall, the case study demonstrates the feasibility of a whitelisting approach for identifying malicious network activity while maintaining a low false positive rate.

المنارة للاستشارات
www.manaraa.com

CHAPTER 7

LIMITATIONS

7.1 Overview

Some limitations of each system presented in thesis were discussed earlier in the context of defining contributions. This chapter describes more precisely what security threats are handled, and what type of information can be protected. While this thesis describes a number of novel security mechanisms, they are by no means fool-proof or computationally free. Like any security system, they can be attacked, but they make the adversary's job much more difficult.

This chapter covers the limitations of each system from the previous chapters one by one, and suggests some improvements. Each of the systems is designed to combat a different set of threats and protect various types of information. Overall, the Storage Capsule system is the most robust against malicious software, but is limited because it can only protect local files. The formatting and timing filters for detecting network security threats are great at differentiating legitimate applications from one another, but are vulnerable to spoofing by malicious programs. The leak measurement algorithms have the advantage of also being able to detect insider leaks. However, some entropy remains in outbound web traffic that malware could use to leak small amounts of data.

7.2 Storage Capsule Limitations

Storage Capsules prevent malicious software from being able to steal information from protected local files. They achieve this goal by cutting off all network and device output during secure mode. Any changes that the primary operating system makes to sensitive files in secure mode are re-encrypted by a trusted component. When the user is finished accessing a file, all changes other than those to the storage capsule are discarded.

المنارة للاستشارات
www.manaraa.com

7.2.1 Security Limitations

The Capsule system has some security limitations. First, it relies on trusted components, including the virtual machine monitor (VMM), the secure virtual machine (VM), the computer hardware, and the user. If any of these elements becomes compromised, then all bets are off. However, we believe that this is a reasonable threat model because the trusted components are much more secure than a standard operating system. Virtual machine monitors contain far less code and experience a much lower rate of vulnerabilities [Secunia09a] than commodity OS kernels [Secunia09b]. The secure VM in Capsule runs only a very limited service to support Storage Capsules, and everything else is blocked by a firewall. Finally, attacking hardware and individuals carries greater risks because it requires physical interaction.

Another attack on the Capsule system involves malware tricking the user. Even if the user is not malicious, a gullible user can be just as bad. The Capsule system asks the user to enter a key escape during every transition to secure mode. Malware could skip this step and spoof the password entry UI to steal the decryption password. If the user forgets about the escape sequence, then Capsule cannot stop this attack. The primary countermeasure to spoofing is user education. It may also be beneficial in the future to dedicate a portion of the screen to displaying the security level to make the transition process more clear to the user.

Assuming that the trusted Capsule components maintain their integrity, malicious software can still extract data from a protected Storage Capsule under some circumstances. In particular, there are a number of covert channels in the Storage Capsule system. Chapter 3, section 5 discusses the nature of several covert channels in greater detail. Of these, we believe that the following covert channels exist in the current implementation of the Capsule system, which uses VMWare workstation and does not over-commit memory to virtual machines. This is not necessarily a complete list of covert channels, and measuring the bandwidth of these channels is future work.

- **Transition Timing** – The primary VM has a small amount of control over the transition timing going back to normal mode because it can delay the system for up to 30 seconds.
- **Social Engineering** – Malware in the primary OS can instruct the user to perform some inane task upon returning to normal mode that actually leaks a few bits of information. The only defense against this line of attack is educating users.
- **CPU State –** The CPU holds a great deal of state that helps optimize execution speed, such as in caches and branch prediction tables. Malware may be able to manipulate this state during secure mode and read it back by executing time-sensitive operations in normal mode. This covert channel is quite noisy, however, because the VMM executes a

www.manaraa.com

lot of instructions to revert the primary VM's state before it can resume execution in normal mode.

- **Disk Cache State –** Although all disk writes during secure mode are deleted (and thus have minimal impact on cache state), reads made by the primary VM may affect disk cache contents on the VMM. Again, this is a noisy channel because the VMM accesses a large amount of data on the disk during the restoration and transition process going back to normal mode.

7.2.2 Usability Limitations

The Storage Capsule system is designed to protect files that are edited locally. It cannot provide security in an interactive context, such as an online banking session. Every time the user transitions to secure mode and back to normal mode, a latency of 25-65 seconds occurs. While this may be acceptable for editing some document for several minutes, it would not work for a website session or a chat conversation.

Another limitation of the Capsule system is that it disrupts background tasks. If the user is downloading a file, switching to secure mode to access a Storage Capsule will cut off the download. Any other computation that is running in the background may continue to execute, but its results will be erased if they are not saved in a Storage Capsule. In general, any task that relies on device access or saving data outside of the Storage Capsule will break in the Capsule system. One possible solution to this problem is to fork the primary VM for the transition to secure mode, and allow the original version to continue running in normal mode with disk and device access. However, allowing malware to potentially run in two side-by-side virtual machines opens up a whole host of covert channels that would impact security.

Because the Storage Capsule system runs on end hosts, it has significantly more overhead and less coverage than a network-based system. An administrator must install the Capsule system on every computer that requires protection, which can be quite cumbersome. Furthermore, mobile computers that do not support Storage Capsules may enter and leave the network. Sensitive data on these machines would not be protected by the Capsule system.

7.3 Formatting and Timing Limitations

The formatting filters described in Chapter 4 will look at the content of each outbound web request and compare its format to that of an expected browser request. Verifying formatting in this manner has some limitations for both malicious and legitimate applications. First,

المنارة للاستشارات
www.manaraa.com

malicious applications can easily emulate the formatting of a browser by simply copying its headers. Many malicious requests do have different formatting, either because attackers do not know any better or made a mistake. However, nothing fundamentally prevents them from avoiding detection by spoofing web browser request format. Legitimate programs do not necessarily try to spoof a browser, but do not always make requests with identifiable formatting. This occurs most often when almost all HTTP headers are omitted, leaving no unique format. It can also occur when a program calls a standard library that sets the entire HTTP request formatting without indicating the program making the request.

The timing filters analyze the regularity and delay between HTTP requests made to a particular server. The goal is to identify non-human request patterns. Presumably, network applications making web requests will set off these filters. These filters are limited in a few ways. First, an application has to have frequent enough activity to trip the regularity filter. If some spyware program extracts data from a computer over a half-hour period and then goes away, the filter cannot differentiate it from human web browsing. Second, programs that want to remain active over a long period of time can evade the filters by piggybacking on user activity. This allows them to closely mimic the timing of human web browsing and avoid detection.

7.4 Leak Quantification Limitations

The algorithms presented in Chapter 5 determine a tight upper bound on the amount of information in outbound web traffic. This information can be invaluable for forensic analysis. With some improvements, these algorithms can also play an important role in an intrusion detection system. However, the leak quantification techniques have some limitations. First and foremost, they measure the *maximum* amount of information that could be leaking in network traffic (i.e. its entropy), but do not say anything about the size of *actual* leaks. This is because the analysis engine runs at the network level and cannot tell the difference between data that is random and data that is crafted to convey information. An example of this limitation is a request that includes a random number in the path to prevent caching. From the outside, one cannot tell whether this number actually came from the random number generator, or if malware has manipulated the value. The bandwidth of such entropy channels is much smaller than overall bandwidth, but it is enough to leak small pieces of information, like individual names and passwords, without being detected.

If used for intrusion detection, the leak measurement algorithms suffer from another problem: legitimate outbound traffic with a high bandwidth. If users are in the habit of uploading

المنارة للاستشارات
www.manaraa.com

large files, which the algorithms must count, then it will be hard to establish a low threshold for normal activity. One approach for dealing with this problem is to separate file uploads from other unconstrained outbound bandwidth. By putting files in their own bin, it is easier for a security analyst to inspect and process the files with a separate tool, such as a data loss prevention system. Also, sequestering file uploads makes it possible to tighten the threshold on other outbound traffic, thus minimizing the size of potential leaks.

Another major limitation of the current implementation is performance. The leak measurement algorithms can only process traffic at about 2 Mbps. There are several factors that contribute to this low maximum processing speed. The most costly part of the analysis is edit distance computation. Some of the strings are a few thousand characters long, and edit distance is an **O(N^2)** algorithm. There are a few ways to speed up edit distance computations. First, they can be approximated by chopping strings up into multi-character blocks instead of single-character blocks. By considering 4-character chunks, the number of comparisons would be reduced to 1/16, and the cost of each comparison would increase by 4 times, yielding 1/4 of the original cost. Perhaps the most promising optimization for edit distance would be to employ hardware acceleration. FPGAs are relatively inexpensive and could fit into an intrusion detection appliance. Given enough space on the chip, hardware can calculate edit distance in **O(N)** clock cycles, and can perform the calculation in parallel for multiple pairs of strings.

While edit distance is the most expensive part of the leak measurement algorithms, webpage and Javascript processing can also be costly. Computing the precise unconstrained bandwidth requires analyzing every HTML page and Javascript file returned by each server. Because page analysis is fully parallelizable per page, one way to increase throughput would be to distribute page analysis over several CPUs or other machines. Another optimization would be to cache analysis results. At the most basic level, pages that are exactly the same do not have to be analyzed again. Many web pages have dynamic content, but the content might not affect the bandwidth measurement results. For example, if a page displays a user's logged-in name, then caching will not impact accuracy. In a production system, one would need to find a balance between cache lifetime and measurement accuracy.

7.5 Whitelisting Limitations

The whitelisting system described in Chapter 6 is able to infer malicious activity by comparing formatting, timing, and bandwidth alerts to a list of allowed behavior. From a security perspective, the whitelisting system inherits the intersection of its individual components'

المنارة للاستشارات
www.manaraa.com

vulnerabilities. If malware is able to avoid triggering an alert for formatting, timing, and bandwidth filters, then it will slip by undetected. Even when malware or a legitimate network application does trigger an alert, however, there are some circumstances under which it may be able to escape identification by the whitelist.

The only way for malware that triggers an alert to avoid detection is for that alert to match some legitimate behavior on the whitelist. One prime example of such an attack would be a web mail tunnel. If malware connects to an allowed web mail service while emulating web browser request formatting, then it may be able to avoid detection. The key here is that there may be whitelist entries for the web mail service that allow timer-driven or high-bandwidth requests. Malware could exploit this knowledge by blending in with more regular and higher-bandwidth legitimate traffic. The flip side of funneling illicit communication through a legitimate site, however, is that the website owner can track and shut down suspicious accounts. One way to combat this type of mimicry attack would be to reduce the number of applications on the whitelist and the permissiveness of their entries.

Some legitimate applications are also hard to identify with the whitelist because they do not generate unique alerts. Imagine a program that makes requests with no user-agent value or headers to many arbitrary servers. We cannot add a whitelist entry to ignore formatting alerts to all servers, but not doing so leads to false positives. In the two-year test deployment, we witnessed one such class of applications that exhibited this behavior: RSS readers. They can make requests without a user-agent to any server that hosts an RSS feed. In this particular case, we had to add a special filter to the whitelisting system that inspected response content. If a request is made without a user-agent that generates a response that is an RSS XML document, then the alert is associated with the RSS reader application and ignored. In the future, it may be necessary to add other special content filters to differentiate traffic from legitimate applications that use generic formatting.

Finally, whitelisting is limited in that it only detects, and does not prevent bad behavior. Due to its susceptibility to false positives from new applications and versions, blocking everything that does not match the whitelist would have an impact on the functionality of network applications. Detecting bad behavior after the fact means that malware may have already stolen some sensitive information.

المنارة للاستشارات

www.manaraa.com

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Contributions

This thesis introduces novel methods for protecting sensitive information from malicious software. It makes the following contributions:

- **Storage Capsules** – Protect sensitive files while they are accessed in plain text on the end host.
- **Network-Based Threat Detector** – Identifies web requests from different network applications, including malware.
- **Information Leak Quantification** – Measures the maximum amount of information that can be leaked in outbound web traffic.
- **Whitelist for Allowed Web Traffic –** Filters web traffic from known good network applications, leaving behind suspicious activity.

Storage Capsules are a new mechanism for securing files on a personal computer. They are similar to existing encrypted file containers, but also protect information when it is decrypted. Storage Capsules prevent all direct information leaks by an infected primary operating system or application that accesses data during a secure mode. The Capsule system also cuts off many covert channels. The major benefit of Storage Capsules over other security systems is that they are compatible with existing operating systems and applications. Storage Capsules only add a small overhead during secure mode, increasing time for the Apache build benchmark by 38% compared to a native machine without virtualization. Transitions to and from secure mode are also reasonable, taking 4.5 seconds to enter secure mode and 20 seconds to revert to normal mode for a system with 512 MB of RAM. Storage Capsules fulfill their goal of protecting sensitive files, while having only a small impact on performance and the user's workflow.

A major part of the Storage Capsule research was examining potential covert channels. We looked at the hardware layer, the VMM layer, external devices, and in the Capsule system

المنارة للاستشارات
www.manaraa.com

itself for places that an infected primary VM could hide data and recover it outside of secure mode. This research explored covert channels from a new point of view because Storage Capsules have a different threat model. Instead of two compromised processes running side-by-side, one runs after the other has terminated and been erased. This makes it harder, but not impossible, for the adversary to exploit covert channels. We identified some traditional covert channels that may exist in Capsule, such as CPU state and disk cache state. This research also identifies previously unexplored covert channels that are created by memory optimizations at the VMM layer. The Capsule system makes an effort to mitigate as many covert channels as possible and minimize the potential for malicious software to steal sensitive data.

The next part of this thesis focuses on achieving security at the network level. While network security systems have less visibility and control than host-based security systems, they benefit from more complete coverage, easier deployment, and additional resilience to attack. In practice, it is best to deploy multiple layers of security. In the context of this thesis, the network monitoring systems help protect information that cannot fit into Storage Capsules, such as website passwords. They can also provide security for computers that either cannot or are not running Storage Capsule software.

The first networking monitoring system presented in this thesis, Web Tap, examines outbound web traffic. It uses novel techniques for examining both the formatting and timing of web requests. The request headers and protocol fields are checked against specifications to detect different implementations of the HTTP protocol. The resulting alerts indicate the presence of different applications. Non-human request timing characteristics can also indicate the presence of network applications. One method of identifying such activity is to measure the delay between requests for the same site and examine the cumulative distribution of the sorted delay vector. This vector contains clear jumps when requests are running on a timer. Another timing analysis method counts the regularity of intervals that contain activity, as well as the bandwidth variation within those intervals. This helps to detect unusually frequent activity that is not timer-driven, such as spyware reporting every website that a user visits. Finally, we looked at the time of day during which requests occurred to isolate requests that did not come from human activity.

The timing and formatting web traffic filters were evaluated against web browsing data from 30 users over a 40-day time period, as well as against a specially constructed tunneling program. During the 40 days, there were a total of 623 alerts, 45 of which were false positives. This equates to 1.13 false positives per day for 30 users, which is a reasonably low number. Web Tap was also able to detect a specially constructed tunneling program. The tunneling program impersonated the format of a web browser, made periodic callbacks, executed shell commands,

المنارة للاستشارات
www.manaraa.com

and transferred files. Web Tap detected the tunneling program in about 7 hours, even if no commands were executed and no data was sent.

This thesis next examined information leak quantification for outbound web traffic. Raw web traffic contains a lot of data, and is a great place for malicious software to hide information. The goal of leak measurement algorithms in this thesis was to discard repeated and constrained bytes in legitimate web traffic, thus differentiating information leaks from benign activity. Previously, the best method of doing this was to use a standard compression algorithm. However, this naïve approach misses a lot of constrained data due to lack of protocol knowledge. We processed each request and response with an analysis engine that is aware of HTTP, HTML, and Javascript interactions. It extracts link URLs, form fields, and cookies from web server responses. It also looks at the content of requests to determine expected header values for future requests. The underlying principle of the leak quantification techniques is to create a library of expected requests, and then compute the edit distance between actual requests and those in the library. This yields an upper bound on the amount of original information that can be present in web requests.

The leak quantification engine was evaluated on controlled browsing scenarios, as well as real traffic from ten users over a 30-day period. Only data from the controlled scenarios was used to develop the engine; no modifications were made in response to observations from the real traffic. In the best case, where the leak measurement algorithms were able to correctly identify all links in a browsing scenario, we were able to discount 99.7% of the raw traffic because it was constrained by the protocol, leaving just 1.9 bytes of information per request. In the worst controlled scenario, we could discount 98.9% of raw traffic, leaving 14.5 bytes per request. For the real web browsing traffic, volunteers used various types of browsers, and some cached objects were unavailable to the analysis engine. Even so, we were able to discount 98.5% of the overall raw bandwidth. A standard compression algorithm was only able to compress away 90.1% of the data, an order of magnitude worse than our results. This thesis showed that most data in outbound web requests is constrained by the protocol specifications and can be discarded when searching for information leaks.

Malicious programs are no fundamentally different than legitimate applications when looking at their network traffic. The systems described earlier in this thesis examine timing, formatting, and request bandwidth, and can only say that traffic is coming from some network application. They cannot say if an application has malicious intent. For this purpose, we employ a whitelist. Each entry in the whitelist specifies a type of alert – timing, formatting, or bandwidth – as well as an optional time, client, or server mask. Each whitelist entry describes the way in which a program's network behavior differs from that of a standard web browser. This not only

المنارة للاستشارات
www.manaraa.com

allows us to identify known unwanted programs that are running in a network, but also infer that alerts not matching a whitelist entry are suspicious.

The web activity whitelist was evaluated by deploying it in a network with over 700 computers for a two-year period. This particular network had relatively open policies, meaning that users could install any legitimate application they wished. At first, there was a higher rate of new whitelist entries that peaked at 350 per month, or approximately 12 per day. As time went on and the whitelist became more mature, the rate of new whitelist entries declined to 50 per month – less than two per day. This deployment showed that whitelisting has a reasonable overhead in a live environment with a diverse set of programs and operating systems. The whitelist was able to successfully identify a wide variety of network applications, and led to the discovery of numerous security threats.

8.2 Future Work

Every research project that solves an important problem stumbles upon two more. Here are some of the key areas for future work that have come about as a result of this thesis:

- **Further Investigation of Covert Channels:** Chapter 3 describes several covert channels through which malicious software could leak information in the Capsule system. However, we are not sure how to most effectively exploit these covert channels and have not measured their bandwidth. Implementing processes that transfer information via the covert channels described in Chapter 3 would further the knowledge and understanding of how to mitigate covert channels.
- **Usability Study of Key Escape Sequences:** Storage Capsules rely on asking the user to enter keyboard escape sequences to securely transition between modes. It is unclear how susceptible the average user would be to an attack where malware asked the user to enter a file decryption password without first prompting for the escape sequence. It is also unknown how effective education would be in mitigating such an attack. A usability study on key escape sequences would help us understand how effective this mechanism is in production systems.
- **Browsing Session Tracking:** Theoretically, a computer should only request a URL that does not come from a link on a previous website when it is manually entered by the user. Each request that does not come from a link on an open page is part of a new browsing session initiated by the user, and should be subject to higher scrutiny. For example, spyware might send a request to its home server that contains some private information.

المنارة للاستشارات
www.manaraa.com

This request would start a new browsing session, and should be considered suspicious because it not a common or trusted site. Browsing session tracking has the potential to help pinpoint spyware activity.

- **Quantifying Leaks in Other Protocols:** The current leak measurement algorithms focus on HTTP. In the future, similar techniques can be applied to other protocols. E-mail (SMTP) would be the most challenging because a majority of its data is free-form. File attachments, however, will have their own structure that comes from the application that created them. Other protocols and file formats are likely to have varying levels of entropy that can be filtered out with techniques similar to those used for HTTP.
- **Dynamic Script Analysis with Input Hints:** Right now, the leak measurement engine only executes Javascript that runs at page load time. Some scripts will dynamically construct link URLs or make AJAX requests in response to user input, which we cannot process with the current architecture. One option would be to speculatively generate possible user input events, but this would lead to an exponential blow-up in processing time. Instead, an agent that runs on each client could capture and record the actual events that are sent to Javascript and forward them to the leak measurement engine. Then, it could extract dynamic link URLs and only count the size of human input events.
- **Integrate Leak Quantification with an IDS:** Right now the leak quantification methods presented in Chapter 5 are primarily useful for forensic analysis. In the future, they could be integrated into an intrusion detection system. Before this can happen, research must take place to determine appropriate thresholds for typical unconstrained outbound bandwidth. Then, computers that violate these thresholds can be flagged for further investigation. Performance optimizations would also be necessary for the leak quantification techniques to handle traffic for a medium- to large-sized network. These optimizations would likely include hardware support for edit distance calculation and caching of Javascript and web page processing results.
- **Multi-Organization Whitelist Deployment:** The whitelisting system in Chapter 6 was evaluated in a single enterprise network. We witnessed some overlap between applications running on different computers in that network. However, it is unclear how the set of programs in the test network compares to the overall distribution of software across all enterprises. Deploying whitelist-based detection systems in a variety of enterprise networks with different security policies would paint a better picture of the overall diversity and distribution of network applications.

المنارة للاستشارات
www.manaraa.com

BIBLIOGRAPHY

[Abad01] C. Abad. IP Checksum Covert Channels and Selected Hash Collision. Technical Report, 2001.

[Adobe09] Adobe Systems Incorporated. Adobe Flash Player. http://www.macromedia.com/software/flash/about, Feb. 2009.

[Ahsan00] K. Ahsan. Covert Channel Analysis and Data Hiding in TCP/IP. Master's Thesis, University of Toronto, 2000.

[Ahsan02] K. Ahsan and D. Kundur. Practical Data Hiding in TCP/IP. *Proceedings of the ACM Workshop on Multimedia Security*, Dec. 2002.

[Anderson98] R. Anderson and F. Petitcolas. On the Limits of Steganography. *IEEE Journal of Selected Areas in Communications*, 16(4):474-481, 1998.

[AP05] The Associated Press. DSW Data Theft Much Worse Than Predicted. *USATODAY*, http://www.usatoday.com/tech/news/computersecurity/infotheft/2005-04-19-credit-card-dsw_x.htm, Apr. 19 2005.

[Arbor09] Arbor Networks. Peakflow X. http://www.arbornetworks.com/en/peakflow-x.html, Feb. 2009.

[Axelsson00] S. Axelsson. The Base-rate Fallacy and the Difficulty of Intrusion Detection. *ACM Transactions on Information and System Security*, 3(3):186–205, Aug. 2000.

[Barford98] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *BU Computer Science Technical Report*, BUCS-TR-1998-023, 1998.

[Bell75] D. Bell and L. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. *Technical Report MTR-2997, Mitre Corporation*, Bedford, MA, 1975.

[Biba75] K. Biba. Integrity Considerations for Secure Computer Systems. *Technical Report MTR-3153, Mitre Corporation*, Bedford, MA, Jun. 1975.

[Blaze93] M. Blaze. A Cryptographic File System for UNIX. *Proceedings of the 1st ACM Conference on Computer and Communications Security*, Fairfax, VA, 1993.

المنارة للاستشارات
www.manaraa.com

[Boebert85] W. Boebert and R. Kain. A Practical Alternative to Hierarchical Integrity Policies. *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, MD, 1985.

[Borders04] K. Borders and A. Prakash. Web Tap: Detecting Covert Web Traffic. *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2004.

[Borders06] K. Borders, X. Zhao, and A. Prakash. Securing Sensitive Content in a View-Only File System. *Proceedings of the 6th ACM Workshop on Digital Rights Management*, Oct. 2006.

[Borders07] K. Borders, A. Prakash, M. Zielinski. Spector: Automatically Analyzing Shell Code. *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, Dec. 2007.

[Brand85] Sheila L. Brand. DoD 5200.28-STD Department of Defense Trusted Computer System Evaluation Criteria (Orange Book). National Computer Security Center, Dec. 1985.

[Browne94] R. Browne. An Entropy Conservation Law for Testing the Completeness of Covert Channel Analysis. *Proceedings of the 2nd ACM Conference on Computer and Communication Security (CCS)*, Nov. 1994.

[Brumley06] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-based Signatures. *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[Brumley07] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. *Proceedings of the 16th USENIX Security Symposium*, Boston, MA, Aug. 2007.

[Bugnion97] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.

[Caballero07] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2007.

[Cabuk04] S. Cabuk, C. Brodley, and C. Shields. IP Covert Timing Channels: Design and Detection. *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2004.

[Castro06] S. Castro. How to Cook a Covert Channel. *hakin9*, http://www.grayworld.net/projects/cooking_channels/hakin9_cooking_channels_en.pdf, Jan. 2006.

www.manaraa.com

[Chen01] P. Chen and B. Noble. When Virtual is Better than Real. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[Chow04] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[Christodorescu05] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware Malware Detection. *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[Cid09] D. Cid. OSSEC Open Source Host-based Intrusion Detection System. http://www.ossec.net/, Feb. 2009.

[Czeskis08] A. Czeskis, D. St. Hilair, K. Koscher, S. Gribble, and T. Kohno. Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications. *Proceedings of the 3rd USENIX Workshop on Hot Topics in Security (HOTSEC '08)*, Aug. 2008.

[Danzig92] P. Danzig, S. Jamin, R. C´aceres, D. Mitzel, and D. Estrin. An Empirical Workload Model for Driving Wide-area TCP/IP Network Simulations. *Internetworking: Research and Experience*, 3(1):1– 26, 1992.

[Delio04] M. Delio. Linux: Fewer Bugs than Rivals. *Wired Magazine*, http://www.wired.com/software/coolapps/news/2004/12/66022, Dec. 2004.

[Denning76] D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Denning77] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, Jul. 1977.

[Dingledine04] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-generation Onion Router. *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

[Dunlap02] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[Duska97] B. Duska, D. Marwood, and M. J. Feeley. The Measured Access Characteristics of World Wide Web Client Proxy Caches. *Proceedings of USENIX Symposium on Internet Technology and Systems*, Dec. 1997.

[Dyatlov03] A. Dyatlov and S. Castro. Exploitation of Data Streams Authorized by a Network Access Control System for Arbitrary Data Transfers: Tunneling and Covert Channels Over the HTTP Protocol. http://gray-world.net/projects/papers/covert_paper.txt, Jun. 2003.

المنارة للاستشارات
www.manaraa.com

[Dyatlov09a] A. Dyatlov and S. Castro. Wsh 'Web Shell'. http://www.gray-world.net/pr_wsh.shtml, Feb. 2009.

[Dyatlov09b] A. Dyatlov. Firepass. http://www.gray-world.net/pr_firepass.shtml, Feb. 2009.

[Feng03] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[Fenton74] J. Fenton. Memoryless Subsytems. *The Computer Journal*, 17(2):143–147, May 1974.

[Fielding99] R. Fielding, J. Gettys, J. Moful, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hyptertext Transfer Protocol – HTTP/1.1. *Request for Comments (RFC) 2616*, http://www.w3.org/Protocols/rfc2616/rfc2616.html, Jun. 1999.

[Fisk02] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil. Eliminating Steganography in Internet Traffic with Active Wardens. *Proceedings of the 5th International Workshop on Information Hiding*, Oct. 2002.

[Foster02] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive Type Qualifiers. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, Jun. 2002.

[Fruhwirth09] C. Fruhwirth. LUKS – Linux Unified Key Setup. http://code.google.com/p/cryptsetup/, Feb. 2009.

[Gailly08] J. Gailly and M. Adler. The gzip Home Page. http://www.gzip.org/, Feb. 2009.

[Garfinkel03a] T. Garfinkel, B. Pfaff, J, Chow, M. Rosenblum, D. Boneh. Terra: a Virtual Machine-based Platform for Trusted Computing. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[Garfinkel03b] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. *Proceedings of the10th ISOC Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, Feb. 2003.

[Garfinkel03c] T. Garfinkel and M. Rosenblum. A Virtual Machine Intropsection Based Architecture for Intrusion Detection. *Proceedings of the 10th ISOC Symposium on Network and Distributed System Security (NDSS),* San Diego, CA, Feb. 2003.

[Garfinkel05] T. Garfinkel and M. Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Jun. 2005.

www.manaraa.com

[Gat76] I. Gat and H. Saal. Memoryless Execution: A Programmer's Viewpoint. *Software: Practice and Experience*, 6(4):463–471, 1976.

[Giles03] J. Giles and B. Hajek. An Information-theoretic and Game-theoretic Study of Timing Channels. *IEEE Transactions on Information Theory*, 48:2455–2477, Sep. 2003.

[Gligor93] V. Gligor. A Guide to Understanding Covert Channel Analysis of Trusted Systems. *National Computer Security Center Technical Report*, NCSC-TG-030, Ft. George G. Meade, MD, Nov. 1993.

[Gold79] B. Gold, R. Linde, R. Peeler, M. Schaefer, J. Scheid, and P. Ward. A Security Retrofit of VM/370. *AFIPS Proceedings, 1979 National Computer Conference*, 1979.

[Goldberg74] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pp. 34–35, June 1974.

[Govil99] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors. *Proceedings of the Symposium on Operating System Principles*, Dec. 1999.

[Gray94] J. Gray III. Countermeasures and Tradeoffs for a Class of Covert Timing Channels. *Hong Kong University of Science and Technology Technical report*, 1994.

[Gu08] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, Jul. 2008.

[Handley01] M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. *Proceedings of the 10th USENIX Security Symposium*, Aug. 2001.

[Halder05] V. Haldar, D. Chandra, and M. Franz. Practical, Dynamic Information Flow for Virtual Machines. *Proceedings of the 2nd International Workshop on Programming Language Interference and Dependence*, London, UK, Sep. 2005.

[Halderman08] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. *Proceedings of 17th USENIX Security Symposium*, Jul. 2008.

[Heinz04] F. Heinz, J. Oster. Nstxd – IP Over DNS Tunneling Daemon. http://www.digipedia.pl/man/nstxd.7.html, Mar. 2005.

[Hofmeyr98] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6:151–180, 1998.

www.manaraa.com

[Hopster09] Hopster. Bypass Firewall – Bypass Proxy – HTTP Tunnel Software. http://www.hopster.com/, Feb. 2009.

[Huffman52] D. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, 1952.

[IBM09a] IBM Internet Security Systems (ISS). RealSecure Network Gigabit. http://www-935.ibm.com/services/us/index.wss/offering/iss/a1026965, Feb. 2009.

[IBM09b] IBM Internet Security Systems (ISS). Proventia Network Intrusion Prevention System. http://www-935.ibm.com/services/us/index.wss/offerfamily/iss/a1030570, Feb. 2009.

[IEEE08] IEEE Computer Society. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2007*, Apr. 2008.

[Jaeger03] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C., Aug. 2003.

[Joshi05] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. *Proceedings of the Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, Oct. 2005.

[Kang95] M. Kang, I. Moskowitz, and D. Lee. A Network Version of the Pump. *Proceedings of the 1995 IEEE Symposium in Security and Privacy*, May 1995.

[Katayama03] F. Katayama. Hacker hits up to 8M credit cards. *CNN*, http://money.cnn.com/2003/02/18/technology/creditcards/, Feb. 27, 2003.

[Kelly02] T. Kelly. Thin-Client Web Access Patterns: Measurements From a Cache-busting Proxy. *Computer Communications*, 25(4):357–366, Mar. 2002.

[Kemmerer83] R. Kemmerer. An Approach to Identifying Storage and Timing Channels. *ACM Transactions on Computer Systems*, 1(3), Aug. 1983.

[Kim94] G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. *Proceedings of the 2nd AVM Conference on Computer and Communications Security (CCS)*, Fairfax, VA, Nov. 1994.

[King05] S. King, P. Chen. Backtracking Intrusions. *ACM Transactions on Computer Systems (TOCS)*, 23(1):51-76, Feb. 2005.

[Kocher96] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *Advances in Cryptology: Proceedings of CRYPTO '96*. Aug. 1996.

المنارة للاستشارات

www.manaraa.com

[Kruegel02] C. Kruegel, T. Toth and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. *Proceedings of the Symposium on Applied Computing (SAC)*, Spain, Mar. 2002.

[Kruegel03] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2003.

[Lagerkvist08] O. Lagerkvist. ImDisk Virtual Disk Driver. http://www.ltr-data.se/opencode.html#ImDisk, Dec. 2008.

[Liskov02] M. Liskov, R. Rivest, and D. Wagner. Tweakable Block Ciphers. In *Advances in Cryptology – CRYPTO '02*, Aug. 2002.

[Malan00] G. Malan, D.Watson, F. Jahanian, and P. Howell. Transport and Application Protocol Scrubbing. *Proceedings of the IEEE INFOCOM 2000 Conference*, Mar. 2000.

[Mannan05] M. Mannan and P. van Oorschot. On Instant Messaging Worms, Analysis and Countermeasures. *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, Washington, DC, Oct. 2005.

[McAfee09] McAfee, Inc. Antivirus Software and Intrusion Prevention Solutions. http://www.mcafee.com/us/, Feb. 2009.

[McCamant08] S. McCamant and M. Ernst. Quantitative Information Flow as Network Flow Capacity. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tuscon, AZ, Jun. 2008.

[McHugh95] J. McHugh. Covert Channel Analysis. Technical Report, Dec. 1995.

[Meushaw00] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. http://www.vmware.com/pdf/TechTrendNotes.pdf, 2000.

[Microsoft09] Microsoft Corporation. BitLocker Drive Encryption: Technical Overview. http://technet.microsoft.com/en-us/library/cc732774.aspx, Feb. 2009.

[Morse05] K. Morse. Compression Tools Compared. *Linux Journal*, 2005(137):3, Sep. 2005.

[Moskowitz94] I. Moskowitz and A. Miller. Simple Timing Channels. *Proceedings of the IEEE Symposium on Security and Privacy*, May 1994.

[Mozilla09a] Mozilla. The Firefox Web Browser. http://www.mozilla.com/firefox/, Feb. 2009.

[Mozilla09b] Mozilla. SpiderMonkey (Javscript-C) Engine. http://www.mozilla.org/js/spidermonkey/, Feb. 2009.

المنارة للاستشارات

www.manaraa.com

[Mukherjee94] B. Mukherjee, L. Heberlein, and K. Levitt. Network Intrusion Detection. *Network, IEEE*, 8(3):26–41, 1994.

[Myers97] A. Myers and B. Liskov. A Decentralized Model for Information Flow Control. *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Santi-Malo, France, 1997.

[Myers99] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, San Antonio, TX, Jan. 1999.

[Myers01] A. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. http://www.cs.cornell.edu/jif, Jul. 2001.

[Netwitness09] NetWitness Corporation. NetWitness – Total Network Knowledge. http://www.netwitness.com, Feb. 2009.

[Newsome05a] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Proceedings of the 12th ISOC Symposium on Network and Distributed System Security (NDSS)*, San diego, CA, Feb. 2005.

[Newsome05b] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[Nguyen-Tuong05] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. *Proceedings of the 20th IFIP International Information Security Conference*, Makuhari Messe, Chiba, Japan, Jun. 2005.

[Niksic98] H. Niksic. GNU Wget. – The Noninteractive Downloading Utility. http://www.gnu.org/software/wget/, Sep. 1998.

[NSA09] National Security Agency. Security-enhanced Linux. http://www.nsa.gov/selinux, Feb. 2009.

[Oberheide07] J. Oberheide, E. Cookie, and F. Jahanian. Rethinking Antivirus: Executable Analysis in the Network Cloud. *Proceedings of the 2nd USENIX Workshop on Hot Topics in Security (HOTSEC '07)*, Boston, MA, Aug. 2007.

[OpenDNS09] OpenDNS. Features – Content Filtering. http://www.opendns.com/solutions/smb/, Feb. 2009.

[Oscar09] OSCAR Protocol for AOL Instant Messaging. http://dev.aol.com/aim/oscar/, Feb. 2009.

[Pavlov09] Igor Pavlov. LZMA SDK (Software Development Kit). http://www.7-zip.org/sdk.html, Feb. 2009.

المنارة للاستشارات

www.manaraa.com

[Paxson98] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.

[Percival05] C. Percival. Cache Missing for Fun and Profit. *Proceedings of BSDCan 2005*, May 2005.

[Petitcolas99] F. Petitcolas, R. Anderson, and M. Kuhn. Information Hiding – A Survey. *Proceedings of the IEEE*, 87(7):1062-1078, Jul. 1999.

[Polychronakis06] M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-Level Polymorphic Shellcode Detection Using Emulation. *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2006.

[Proctor07] P. Proctor, R. Mogull, and E. Quellet. Magic Quadrant for Content Monitoring and Filtering and Data Loss Prevention. *Gartner RAS Core Research Note*, G00147610, Apr. 2007.

[Provos04] N. Provos. A Virtual Honeypot Framework. *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

[Randazzo04] M. Randazzo, M. Keeeney, E. Kowalski, D. Cappelli, and A. Moore. Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector. *CERT Report by U.S. Secret Service and CERT Coordination Center*, http://www.secretservice.gov/ ntac/its_report_040820.pdf, Aug. 2004.

[Reuters06] Reuters. U.S. Says Personal Data on Millions of Veterans Stolen. *The Washington Post*, http://www.washingtonpost.com/wp-dyn/content/article/2006/05/22/AR2006052200690.html, May 22 2006.

[Richardson07] R. Richardson. CSI Computer Crime and Security Survey. 2007.

[Rissanen79] J. Rissanen and G. Langdon. Arithmetic Coding. *IBM Journal of Research and Development*, 23(2):146–162, Mar. 1979.

[Roesch99] M. Roesch. Snort – Lightweight Intrusion Detection for Networks. *Proceedings of the 13th USENIX Systems Administration Conference (LISA)*, Seattle, WA, 1999.

[Roshal09] A. Roshal. WinRAR Archiver, a Powerful Tool to Process RAR and ZIP Files. http://www.rarlab.com/, Feb. 2009.

[Rowland97] C. Rowland. Covert Channels in the TCP/IP Protocol Suite. *First Monday*, 1996.

[RSA07] RSA Security Inc. RSA Data Loss Prevention Suite – Solutions Brief. http://www.rsa.com/products/EDS/sb/DLPST_SB_1207-lowres.pdf, 2007.

[Rutkowska05] J. Rutkowska. Red Pill... Or How To Detect VMM Using (Almost) One CPU Instruction. http://invisiblethings.org/papers/redpill.html, 2005.

المنارة للاستشارات
www.manaraa.com

[Sailer04] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

[Sandvine09] Sandvine, Inc. Sandvine – Intelligent Broadband Network Management. http://www.sandvine.com, Feb. 2009.

[Schear06] N. Schear, C. Kintana, Q Zhang, and A. Vahdat. Glavlit: Preventing Exfiltration at Wire Speed. *Proceedings of the 5th Workshop on Hot Topics in Networks (HotNets)*, Nov. 2006.

[Secunia09a] Secunia. Xen 3.x – Vulnerability Report. http://secunia.com/product/15863/?task=statistics, Feb. 25th 2009.

[Secunia09b] Secunia. Linux Kernel 2.6.x – Vulnerability Report. http://secunia.com/product/2719/ ?task=statistics, Feb. 25th 2009.

[Servetto01] S. Servetto and M. Vetterli. Communication Using Phantoms: Covert Channels in the Internet. *Proceedings of the IEEE International Symposium on Information Theory*, Jun. 2001.

[Seward07] J. Seward. bzip2 and libbzip2, version 1.0.5 – A Program and Library for Data Compression. http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html, Dec. 2007.

[Shankar06] U. Shankar, T. Jaeger, and R. Sailer. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. *Proceedings of the 13th ISOC Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, Feb. 2006.

[Shannon51] C. Shannon. Prediction and Entropy of Printed English. *Bell System Technical Journal*, 30:50–64, 1951.

[Sun09] Sun Microsystems. Java. http://www.java.com, 2009.

[Symantec09] Symantec Corporation. Spyware Remover: Norton AntiVirus. http://www.symantec.com/norton/antivirus, Feb. 2009.

[TCG06] Trusted Computing Group. Trusted Platform Module Main Specification. http://www.trustedcomputinggroup.org, Ver. 1.2, Rev. 94, June 2006.

[TippingPoint09] TippingPoint Technologies, Inc. TippingPoint Intrusion Prevention Systems. http://www.tippingpoint.com/products_ips.html, Feb. 2009.

[Trostle91] J. Trostle. Multiple Trojan Horse Systems and Covert Channel Analysis. *Proceedings of Computer Security Foundations Workshop IV*, Jun. 1991.

[TrueCrypt09] TrueCrypt Foundation. TrueCrypt – Free Open-Source On-The-Fly Disk Encryption Software. http://www.truecrypt.org/, Feb. 2009.

المنارة للاستشارات
www.manaraa.com

[Vasudevan06] A. Vasudevan and R. Yerraballi. Cobtra: Fine-Grained Malware Analysis Using Stealth Localized-Executions. *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[Vigna04] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-Based Intrusion Detection Signatures Using Mutant Exploits. *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2004.

[Vontu09] Vontu, Inc. Vontu – Data Loss Prevention, Confidential Data Protection. http://www.vontu.com, Feb. 2009.

[Vrable05] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, Dec. 2005.

[Wagner74] R. Wagner and M. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.

[Wagner01] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[Waldspurger02] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[Wang04] K. Wang and S. Stolfo. Anomalous Payload-Based Network Intrusion Detection. *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, France, Sep. 2004.

[Wang06] Z. Wang and R. Lee. Covert and Side Channels Due to Processor Architecture. *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2006.

[Weber07] T. Weber. Criminals 'May Overwhelm the Web'. *BBC News*, http://news.bbc.co.uk/1 /hi/business/6298641.stm, Jan. 25 2007.

[Websense09] Websense, Inc. Web Security Suite. http://www.websense.com/global/en/ProductsServices/WSSecuritySuite/, Feb. 2009.

[Winzip09] WinZip International LLC. WinZip – The Zip File Utility for Windows. http://www.winzip.com/, Feb. 2009.

[Wray91] J. Wray. An Analysis of Covert Timing Channels. *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1991.

[Wright02] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.

المنارة للاستشارات

www.manaraa.com

[XenSource09] XenSource, Inc. Xen Community. http://xen.xensource.com/, Feb. 2009.

[Yumerefenendi07] A. Yumerefendi, B. Mickle, and L. Cox. TightLip: Keeping applications from spilling the beans. *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

[Zhang00] Y. Zhang and V. Paxson. Detecting Backdoors. *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.

[Ziv77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[Zimmerman95] P.R. Zimmermann. The Official PGP User’s Guide. *MIT Press*, 1995.

المنارة للاستشارات

www.manaraa.com