Analyzing Intrusions Using Operating System Level Information Flow

by

Samuel T. King

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
2006

For my wife Sam and my son Eli.

ACKNOWLEDGEMENTS

I would like to thank some of the people who helped me in my journey towards getting a PhD.

First, I would like to thank my PhD advisor, Peter Chen. He was literally an ideal advisor and was the greatest influence in my development as a researcher. We spent countless hours in his office discussing the topics in this dissertation, and these interactions are what made my graduate student life so enjoyable and convinced me to become a faculty member myself.

I would like to thank my committee, Pete, Vineet, Morley, and Brian for their valuable insight and feedback.

I would like to thank Morley and Dom for helping out with some of the multi-host experiments in this dissertation.

I would like to thank the other members of the CoVirt group: Ashlesha, Dom, George, and Murtaza. They helped make research even more fun.

I would like to thank my high school English teacher Mrs. Neagly (seriously). She went well above and beyond what is expected of a school teacher and helped my writing in ways that carry over into my work today.

Finally, I would like to thank my family Sam, Eli, Annie, Milo, Mom, Dad, Traci, Bec, Grammy, Kathie, Dru, Whitney, Bob, Jennie, Chris, and Becca for all of the love and support.

TABLE OF CONTENTS

DEDICATION . . . . **ii**

ACKNOWLEDGEMENTS . . . . **iii**

LIST OF TABLES . . . . **vi**

LIST OF FIGURES . . . . **vii**

CHAPTER

1. Introduction 1

1.1 Background: forensic procedure 4
1.2 Scope of the Dissertation 5
1.3 Dissertation overview 6

2. Information Flow 7

2.1 Background: information flow at different levels of granularity . 8
2.2 Operating-system-level information flow 10

3. Information-Flow Graphs 12

3.1 Objects 12
3.2 Potential dependency-causing events 13
3.2.1 Process/process dependencies 14
3.2.2 Process/file dependencies 15
3.2.3 Process/filename dependencies 15
3.2.4 Process/registry dependencies 16
3.3 Dependency graphs 16
3.3.1 Backward dependency graphs 17
3.3.2 Forward dependency graphs 20
3.4 Dependencies tracked by current prototype 21
3.5 Implementation structure of EventLogger and GraphGen 22

4. Analyzing Intrusions On a Single Host 26

4.1 Prioritizing portions of an information-flow graph 26
4.2 BackTracking attacks 28
4.3 ForwardTracking attacks 37

5. Analyzing Intrusions Across Multiple Hosts 42

5.1 Prioritizing network connections 42
5.2 Tracking multi-hop attacks 44
5.2.1 Slapper worm 45
5.2.2 Multi-vector manual attack 49
5.2.3 E-mail virus 51
5.3 Correlating IDS alerts 52

6. Limitations 56

7. Enhancing Operating-System-Level Information Flow 59

7.1 Course-grained information flow 60
7.2 Advanced OS-level techniques 61
7.3 Fine-grained information flow 62

8. Future Work 64

9. Related Work 66

9.1 Intrusion analysis 66
9.2 Information flow 68
9.3 Multi-hop attacks and correlating IDS alerts 70
9.4 Sandboxing IDSs 70

10. Conclusions 72

BIBLIOGRAPHY **74**

LIST OF TABLES

Table

2.1 Impact of tracking information flow at different levels of granularity. . . 8

4.1 Statistics for BackTracker's analysis of attacks 30

LIST OF FIGURES

Figure

3.1 Constructing a dependency graph 17

3.2 Backward dependency graph for an example with discrete times 19

3.3 System structures for logging events 23

4.1 Mostly-unfiltered dependency graph for *bind attack* 29

4.2 Filtered dependency graph for *bind attack*. 32

4.3 Filtered dependency graph for *ptrace* attack 33

4.4 Filtered dependency graph for *openssl-too attack*. 34

4.5 Filtered dependency graph for *self attack* 36

4.6 Backward information-flow graph for *mslaugh* attack 39

4.7 Forward information-flow graph for *mslaugh* attack 40

5.1 Inter-host propagation of the Slapper worm 46

5.2 Backward information-flow graph for Slapper worm for Host D. 47

5.3 Forward causality graph for the Slapper attack 48

5.4 Inter-host propagation of the multi-vector attack 49

5.5 Tracking an e-mail virus 52

5.6 Correlating alerts 54

CHAPTER 1

Introduction

Modern computer systems are under attack, making computer security an important topic to understand. Malicious people (called hackers or attackers) search for computer systems (hosts) that are vulnerable to attacks. Vulnerable hosts range from personal computers (PCs) used in a home environment to corporate networks that are professionally managed and maintained by system administrators. A computer system becomes a vulnerable host when it contains mistakes or errors that can be exploited by attackers to achieve unauthorized access. For example, an attacker can gain unauthorized access to a PC if a user clicks on a malicious email attachment, or an attacker can gain access to a system if an application running on the host contains a software programming error (vulnerability) that the hacker can exploit. Once an attacker gains access to a system, they can run arbitrary computing tasks on the compromised host.

Recent trends suggest that today's attackers have economic incentives, making attacks likely to continue. Researchers once believed that most hackers were unskilled adolescents motivated primarily by bragging rights [63]. However, in 2004 the USA Today reported [73] that hackers were selling computing time on large networks of compromised machines, called botnets. Botnets are believed to be used by spammers and organized crime syndicates for activities such as online extortion, delivering unsolicited email advertisements, and identity theft through fraudulent web sites. Since hackers can make money by compromising vulnerable host and renting them out, skilled attackers will continue to find new and innovative ways to break into computer systems.

Computer security compromises continue despite current best efforts, providing additional evidence that attacks are likely to continue. In 2005 the F.B.I. conducted a survey of U.S. businesses to learn more about computer security incidents [35]. In this survey, the F.B.I. polled 2066 business and found that 87% of the companies surveyed experienced at least one computer security compromise in 2005. Surprisingly, most of these companies use security countermeasures to prevent compromises. For example, 98% of the companies run antivirus software, 91% use network firewalls, and 75% use antispyware software. This evidence shows that the current state-of-the-art in computer security defenses does not prevent all security compromises.

Successful attacks on computer systems are costly to businesses and to society. For example, in the 2005 Computer Crime Survey [35], the F.B.I. estimates that computer security incidents cost U.S. businesses $67.2 billion a year. Also, recent security incidents have grounded flights [61], brought networks and email systems to a stand still [39], and shut down the Russian stock exchange for several hours [53]. In addition, according to the Los Angeles Times, in the first half of 2006 hackers stole sensitive information (e.g., social security numbers, medical records) from at least 845,000 people associated with universities [28].

Because computer security attacks are costly and likely to continue despite current best efforts, system administrators must be able to perform post-attack analysis of computer intrusions. System administrators must answer two main questions when analyzing intrusions: “how did the attacker gain access to my system?”, and “what did the attacker do after they broke in?”. Determining how an attacker gained access to the systems helps system administrators prevent future intrusions. For example, if a system administrator can determine which application contained the vulnerability that the attacker exploited, they can prevent future intrusions by updating the vulnerable application, or if no update is available, they can disable the application until an update becomes available. Determining the attacker’s activities after they broke in is important because it can help the administrator recover from the intrusion.

Unfortunately, current sources of information used for intrusion analysis suffer from one or more limitations. Host logs are collections of application-level and system-level events. Some host logs contain incomplete information making intrusion analysis difficult. For example, web servers can log http requests. Although web-server logs are rich in semantic information, they provide no system-wide data, making it difficult for administrators to analyze intrusions based solely on web logs. Network-level logs provide a system-wide view of computing activity by recording all network traffic between a host and the network. However, obfuscated network data and undocumented network protocols might complicate intrusion analysis because the system administrator may not be able to infer host states and events based solely on network-level logs. Disk images of infected hosts contain all persistent state resulting from an attack. However, disk images are a snapshot of the persistent state of a system, and may not contain the information needed to determine the transient sequence of states and events that lead to the final disk image. A general limitation of most tools and sources of information is that they intermingle the actions of the intruder (or the state caused by those actions) with the actions/state of legitimate users. Even in cases where the logs and disk state contain enough information to understand an attack, identifying the sequence of events from the initial compromise to the point of detection is still largely a manual process.

We propose using operating-system-level (OS-level) information-flow graphs to highlight the activities of an attacker. OS-level information-flow graphs are a collection of system-level causal events that connect operating system (OS) objects, like processes and files. For example, a process that writes to a file forms a causal link from the process that does the writing to the file that it writes to. These causal events can be linked together to form an information-flow graph which highlights the events and objects that are likely to be part of an attack.

Information-flow graphs are effective at highlighting malicious activities because they hone in on events and objects that are likely to be part of the attack. Intrusion analysis starts with a suspicious object (e.g., malicious process, trojaned executable file) on a system; this suspicious object becomes the starting point for an information-flow graph used to analyze intrusions. Because we start with a malicious object that we know, with a high level of confidence, is part of an attack, it is likely that the sequence of events and objects that affect and are affected by this malicious object are also part of the attack. Focusing on objects and events that are likely to be part of an attack addresses the problem of finding malicious objects and events within audit logs that contains mostly legitimate activities. This key observation is why information-flow graphs are effective at helping system administrators analyze attacks.

We implemented three systems for analyzing intrusions using information-flow graphs. We developed BackTracker to determine how an intruder broke into a system. BackTracker starts with a suspicious object (e.g., email virus) and follows the attack back in time, using causal events, to highlight the sequence of events and objects that lead to the suspicious state. Showing a graph of these causally-connected events and objects provides a system-wide view of the attack and significantly reduces the amount of data an administrator must examine in order to determine which application was originally exploited. We also developed ForwardTracker to determine the attacker's actions after the compromise. ForwardTracker starts from the exploited application and tracks causal events forward in time to display the information flow graph of events and objects that result from the intrusion. Furthermore, we designed and implemented Bi-directional Distributed BackTracker (BDB) that continues the backward and forward information-flow graphs across the network to highlight the set of computers on a local network that are likely to have been compromised by the attacker.

1.1 Background: forensic procedure

Before an administrator can start to understand an intrusion, he or she must first detect that an intrusion has occurred [17]. There are numerous ways to detect a compromise. A tool such as TripWire [44] can detect a modified system file; a network or host firewall can notice a process conducting a port scan or launching a denial-of-service attack; a sandboxing tool can notice a program making disallowed or unusual patterns of system calls [41] [37] or executing foreign code [48]. We use the term *detection point* to refer to the state on the local computer system that alerts the administrator to the intrusion. For example, a detection point could be a deleted, modified, or additional file, or it could be a process that is behaving in an unusual or suspicious manner.

Once an administrator is aware that a computer is compromised, the next step is to analyze the system to recover from the attack [16]. Administrators typically use two main sources of information to find clues about an intrusion: system/network logs and disk state [34]. An administrator might find log entries that show unexpected output from vulnerable applications, deleted or forgotten attack toolkits on disk, or file modification dates which hint at the sequence of events during the intrusion. Many tools exist that make this job easier. For example, Snort can log network traffic; Ethereal can present application-level views of that network traffic; and The Coroner's Toolkit can recover deleted files [33] or summarize the times at which files were last modified, accessed, or created [32] (similar tools are Guidance Software's EnCase, Access Data's Forensic Toolkit, Internal Revenue Services' ILook, and ASR Data's SMART).

Operating-system-level information-flow graphs provide a new source of information to find clues about an intrusion. For the remainder of this dissertation, we discuss how operating-system-level information-flow graphs can help to analyze intrusions.

1.2 Scope of the Dissertation

My thesis is:

Operating-system-level information-flow graphs are effective at helping to analyze intrusions.

We validate this claim by implementing systems that generate information-flow graphs that we use to analyze intrusions. We evaluate these systems using attack data from benchmark workloads where we broke into hosts ourselves, and also from workloads generated from allowing external hackers to break into systems that run our analysis software. Our contributions are:

- Introducing the idea of using information-flow graphs for intrusion analysis.
- Building systems to log causal OS-level events in real time.
- Building systems to generate information-flow graphs based on logs of causal events.
- Developing algorithms for prioritizing portions of information-flow graphs to simplify intrusion analysis.

1.3 Dissertation overview

In this dissertation, we discuss operating-system-level information-flow graphs and how they can be used to help analyze intrusions.

In Chapter 2, we introduce information flow and discuss previous work in the area. We talk about tracking information flow at various levels of granularity, and justify our choice of tracking information flow at the operating system level.

In Chapter 3, we discuss how to build information-flow graphs. We define the objects and events we track, and we show how to use these objects and events to build information-flow graphs.

In Chapter 4, we show how information-flow graphs help to analyze intrusions on local hosts. We describe honeypot experiments where we allow intruders to break into instrumented systems, and we show how backward and forward information-flow graphs help to understand the details of the attacks.

In Chapter 5, we continue the analysis across the network. We discuss network-level causal events and show how they can help track attacks across multiple hosts.

Although we found it useful to analyze intrusions using OS-level information flow, there certainly are limitations. Chapter 6 describes what we believe are the fundamental limitations of our approach.

In Chapter 7, we discuss how to enhance OS-level information flow to reduce the size of large information-flow graphs.

In Chapter 8, we describe possible future directions. This dissertation focuses on intrusion analysis, but information-flow graphs could also be used for intrusion detection and prevention. This chapter discusses the idea of using information-flow graphs for intrusion detection and prevention.

Chapter 9 discusses related work and Chapter 10 concludes.

CHAPTER 2

Information Flow

Information flow is defined as transferring data (information) from one object to another. To track information flow, one must define a set of objects, and identify events where information flows between these objects.

We assume information-flow events are causal events. In other words, actions of an object are assumed to be *affected* by, or caused by, all previous events, even if the the object no longer uses the information gleaned from previous events. This conservative approach can lead to objects in an information-flow graph that do not impact the attack. We use the term *false positive* to describe objects we include in an information-flow graph, but do not actually affect an attack. To reduce false positives, we use filtering rules to prioritize portions of information-flow graphs. This filtering can remove objects that affect the attack and we use the term *false negative* to describe objects that we omit from an information-flow graph, but affect the attack.

Information flow has been studied at many different levels of granularity where each different level of granularity balances performance vs false positives. In general, more coarse-grained information-flow tracking incurs fewer performance penalties, but leads to more false positives. More fine-grained information-flow leads to fewer false positives, but at the cost of performance.

Operating-system-level information flow provides a good balance between performance and false positives. In this chapter we justify this claim by discussing previous projects that track information flow at various levels of granularity. Then, we discuss why we chose OS-level information flow.

	runtime overhead (percent)	***no host instrumentation***	***programming language independent***	***supports legacy applications w/o source***
host level	0%	√	√	√
OS level	9%		√	√
PL level	8%			
instruction level	2500%		√	√

Table 2.1: Impact of tracking information flow at different levels of granularity.

This table summarizes the performance and deployment tradeoffs of using different levels of granularity. We show information flow at the host level, at the operating system level, at the programming language level, and at the instruction level. The factors we show are runtime overhead, need to instrument the host, programming language independence, and legacy application support (i.e., source code not needed). Note: the performance numbers for programming-language level and instruction level information flow only measure the instrumentation overhead. These numbers do *not* include the additional overhead for creating an audit log based on the instrumented events, which is needed for forensic analysis.

2.1 Background: information flow at different levels of granularity

Past projects track information flow at many different levels of granularity including host level, operating-system level, programming-language level, and instruction level. This section discusses some of these past projects with a focus on issues related to the level of granularity being tracked. Section 9.2 describes different uses for information flow.

Table 2.1 shows some of the tradeoffs one must consider when choosing at which level to track information flow. In this table we summarize the tradeoffs of host level, operating-system level, programming-language level, and instruction level information flow.

Host-level information-flow [51, 9, 12, 49] tracks data sent between distinct hosts on a network. Tracking information flow at the host level requires logging only network messages, no instrumentation is needed on the hosts themselves. This can be advantageous for large enterprise networks with diverse host software and multiple distinct administrative domains. Furthermore, current projects have shown how to make this type of logging efficient [12, 49] for enterprise networks. However, analyzing intrusions at the host level of granularity can be challenging because system administrators must infer the actions of individual hosts based solely on network data. For example, if an attacker exploits a user's instant messaging application and installs a backdoor application, connecting the backdoor to the instant messaging application, using only network data, can be difficult. Making this connection is difficult because the administrator must reproduce the semantics of the instant messaging application, including the vulnerability and any non-determinism, using only network messages. In addition, encryption and undocumented network protocols obfuscate network-message semantics and further complicate the forensic procedure when using host-level information flow.

OS-level information flow [14, 2, 45, 47, 40, 79, 76, 30] divides hosts into execution subcomponents (e.g., processes) and state subcomponents (e.g., files). Tracking information flow at this level of granularity gives a system-wide perspective of attack activity and follows data as it travels through the host. This level of granularity adds little runtime overhead (around 9% according to our tests in Section 4.2) and tracks applications independent of programming language and source code availability. However, some OS-level objects tend to accumulate taint which can lead to false positives.

Programming-language-level information flow [26, 27, 55, 77, 75, 74, 8, 69] divides processes into execution subcomponents (i.e., code paths) and state subcomponents (i.e., variables). PL-level information flow tracks efficiently (around 8% according to [75]) data used within a processes. These techniques can determine when to remove taint from a processes and can help reduce many false positives resulting from tracking OS-level information flow alone. However, PL-level techniques assume the use of a particular programming language and require access to source code, which makes PL-level techniques unsuitable for many applications.

Instruction-level information flow [57, 21, 22, 54, 56, 48] divides code paths and variables into execution subcomponents (i.e., instruction streams) and state subcomponents (i.e., memory and CPU registers). Instruction-level information flow operates at the assembly instruction level and does not make any assumptions about programming languages, and does not require source code. However, this flexibility comes at the cost of performance where tracking instruction-level information flow can increase application runtime by several orders of magnitude [57].

2.2 Operating-system-level information flow

When deciding at which level to track information flow, we first examine our requirements for a system suitable for analyzing intrusions. Our requirements are:

- Good performance.
- Support a diverse set of host software.
- Track all applications including foreign code and newly installed code.
- Provide a system-wide perspective of activities on the host with enough detail to understand the attack.
- Provide an information-flow graph that is small enough for visual attack analysis.

OS-level information flow meets all of the above requirements. First, OS-level tracking requires monitoring a relatively small number of objects and events, so we can track these objects and events at runtime inexpensively. Our current prototype adds 9% overhead in our worst-case macro-benchmark experiments (see Section 4.2). Second, the monitored objects and events are common among many of today's operating systems, so these techniques apply to a wide range of host software configurations. We implemented event logging modules for both Linux and Windows operating systems. Third, tracking occurs from within the OS kernel itself, so OS-level information flow monitors all applications automatically without requiring recompilation or *a priori* knowledge of application internals. As a result, OS-level information flow tracks foreign code and newly installed code, which is vital because attackers often install their own binaries on a compromised system. Fourth, OS-level information flow provides a system-wide perspective of an attack by following the flow of information from one object to another. The tracked objects (e.g., processes and files) are familiar to administrators and provide information about the details of an attack. Fifth, OS-level information flow highlights a subset of the total system activity that is likely to be part of the attack being analyzed. Although this subset represents a much smaller set of data, in our experience the resulting information-flow graph tends to be large, making analyzing intrusions difficult. Fortunately, OS-level information-flow graphs are well suited for a number of heuristics which effectively highlight portions of an information-flow graph. This smaller information-flow graph, in our experience, was small enough to effectively analyze intrusions. Chapters 4 and 5 describes the details of the techniques we use to prioritize portions of information-flow graphs.

OS-level information flow is the only level of information flow that meets all of the above requirements. Host-level information flow does not capture enough details about the attack to connect system-level events and objects together, making analyzing intrusions difficult. Programming-language-level information flow does not work for foreign code installed by an attacker because the source code is unavailable generally. Instruction-level information flow adds too much overhead to be practical for enterprise systems. Overall, operating-system-level information flow provides the best balance between all factors for analyzing intrusions.

CHAPTER 3

Information-Flow Graphs

In order to generate information-flow graphs, one must define a set of objects, and track events where information flows between these objects. For this dissertation, we developed *EventLogger* which logs information-flow events in real time, and we built *GraphGen* that processes EventLogger logs to generate the information-flow graphs we use to analyze intrusions.

In this chapter we first discuss the OS-level objects that we track. We then discuss the dependency causing events that must be recorded. Next, we show how these objects and events can be used to create information flow graphs. Finally, we discuss the events and objects tracked by our current prototype and the implementation structure of our system.

3.1 Objects

This section discusses the four types of OS-level objects that we track: processes, files, filenames, and registry entries.

A process is identified uniquely by a process ID and a version number. EventLogger keeps track of a process from the time it is created by a fork or clone system call to the point where it exits. The one process that is not created by fork or clone is the first process (swapper); EventLogger starts keeping track of swapper when it makes its first system call.

A file object includes any data or metadata that is specific to that file, such as its contents, owner, or modification time. A file is identified uniquely by a device, an inode number, and a version number. Because files are identified by inode number rather than by name, we track a file across rename operations and through symbolic links. We treat pipes and named pipes as normal files. Objects associated with System V IPC (messages, shared memory, semaphores) can also be treated as files, though the current EventLogger implementation does not handle these.

A filename object refers to the directory data that maps a name to a file object. A filename object is identified uniquely by a canonical name, which is an absolute pathname with all ./ and ../ links resolved. Note the difference between file and filename objects. In Unix, a single file can appear in multiple places in the filesystem directory structure, so writing a file via one name will affect the data returned when reading the file via the different name. File objects are affected by system calls such as write, whereas filename objects are affected by system calls such as rename, create, and unlink.

A registry object refers a Windows registry entry. Windows uses the registry as a central repository for application and system-wide configuration data. Registry entries include a unique *key* used to identify the object, and also a *value* that represents the data. Unlike filenames and file data, registry key to registry value mappings are unique.

3.2 Potential dependency-causing events

EventLogger logs events at runtime that induce dependency relationships between objects, i.e. events in which one object affects the state of another object. These events are the links that allow us to deduce timelines of events connecting various objects together. A dependency relationship is specified by three parts: a source object, a sink object, and a time interval. For example, the reading of a file by a process causes that process (the sink object) to depend on that file (the source object). We denote a dependency from a source object to a sink object as *source⇒sink*.

We use time intervals to reduce false dependencies. For example, a process that reads a file at time 10 does not depend on writes to the file that occur after time 10. Time is measured in terms of an increasing event counter. Unless otherwise stated, the interval for an event starts when the system call is invoked and ends when the system call returns. A few types of events (such as shared memory accesses) are aggregated into a single event over a longer interval because it is difficult to identify the times of individual events.

There are numerous events which cause objects to affect each other. This section describes potential events that we could track. Section 3.3 describes how we use dependency-causing events. Section 3.4 then describes why some events are more important to track than others and identifies the subset of these dependencies logged by our current prototype. Section 3.5 discusses the implementation structure require to track causality and produce information-flow graphs. We classify dependency-causing events based on the source and sink objects for the dependency they induce: process/process, process/file, process/filename, and process/registry.

3.2.1 Process/process dependencies

The first category of events are those for which one process directly affects the execution of another process. One process can affect another directly by creating it, sharing memory with it, or signaling it. For example, an intruder may login to the system through sshd, then fork a shell process, then fork a process that performs a denial-of-service attack. Processes can also affect each other indirectly (e.g., by writing and reading files), and we describe these types of dependencies in the next two sections.

If a process creates another process, there is a parent⇒child dependency because the parent initiated the existence of the child and because the child's address space is initialized with data from the parent's address space.

Besides the traditional fork system call, Linux supports the clone and vfork system calls, which create a child process that shares the parent's address space (these are essentially kernel threads). Children that are created via clone or vfork have an additional bi-directional parent⇔child dependency with their parent due to their shared address space. In addition, clone and vfork create a bi-directional dependency between the child and other processes that are currently sharing the parent's address space. Because it is difficult to track individual loads and stores to shared memory locations, we group all loads and stores to shared memory into a single event that causes the two processes to depend on each other over a longer time interval. We do this grouping by assuming conservatively that the time interval of the shared-memory dependency lasts from the time the child is created to the time either process exits or replaces its address space through the execve system call.

3.2.2 Process/file dependencies

The second category of events are those for which a process affects or is affected by data or attributes associated with a file. For example, an intruder can edit the password file (process⇒file dependency), then log in using the new password file (file⇒process dependency). Receiving data from a network socket can also be treated as reading a file, although the sending and receiving computers need to cooperate to link the receive event with the corresponding send event.

System calls like write and writev cause a process⇒file dependency. System calls like read, readv, and execve cause a file⇒process dependency.

Files can also be mapped into a process's address space through mmap, then accessed via load/store instructions. As with shared memory between processes, we aggregate mapped-file accesses into a single event, lasting from the time the file is mmap'ed to the time the process exits. This conservative time interval allows us to avoid tracking individual memory operations or the un-mapping or re-mapping of files. The direction of the dependency for mapped files depends on the access permissions used when opening the file: mapping a file read-only causes a file⇒process dependency; mapping a file write-only causes a process⇒file dependency; mapping a file read/write causes a bi-directional process⇔file dependency. When a process is created, it inherits a dependency with each file mapped into its parent's address space.

A process can also affect or be affected by a file's attributes, such as the file's owner, permissions, and modification time. System calls that modify a file's attributes (e.g., chown, chmod, utime) cause a process⇒file dependency. System calls that read file attributes (e.g., fstat) cause a file⇒process dependency. In fact, any system call that specifies a file (e.g., open, chdir, unlink, execve) causes a file⇒process dependency if the filename specified in the call exists, because the return value of that system call depends on the file's owner and permissions.

3.2.3 Process/filename dependencies

The third category of events are those that cause a process to affect or be affected by a filename object. For example, an intruder can delete a configuration file and cause an application to use an insecure default configuration. Or an intruder can swap the names of current and backup password files to cause the system to use out-of-date passwords.

Any system call that includes a filename argument (e.g., open, creat, link, unlink, mkdir, rename, rmdir, stat, chmod) causes a filename⇒process dependency, because the return value of the system call depends on the existence of that filename in the file system directory tree. In addition, the process is affected by all parent directories of the filename (e.g., opening the file /a/b/c depends on the existence of /a and /a/b). A system call that reads a directory causes a filename⇒process dependency for all filenames in that directory.

System calls that modify a filename argument cause a process⇒filename dependency if they succeed. Examples are creat, link, unlink, rename, mkdir, rmdir, and mount.

3.2.4 Process/registry dependencies

The fourth category of events are those for which a process affects or is affected by registry entries. For example, an intruder can edit the HKLM\...\Run registry value (process⇒registry dependency), which causes an attacker-specified process to launch (registry⇒process) each time Windows starts.

Processes access registry values using query, set, and create system calls. Query causes a registry⇒process dependency by reading the value of a registry entry. Set and create cause a process⇒registry dependency by writing the value of a registry entry.

Registry keys are access using create, delete, enum, and open system calls. Create and delete cause a process⇒registry dependencies by creating and deleting keys respectively. Enum and open system calls cause registry⇒process dependencies by enumerating subkeys within a key or by opening existing keys.

3.3 Dependency graphs

By logging objects and dependency-causing events during runtime, we save enough information to build an information-flow graph that depicts the dependency relationships between all objects seen over that execution. Rather than presenting the complete dependency graph, however, we would like to make understanding or representing an attack as easy as possible by presenting only the relevant portion of the graph. This section describes

```
/* read events from latest to earliest */
foreach event E in log {
  foreach object O in graph {
    if(E affects O by the time threshold for object O) {
      if(E's source object not already in graph) {
        add E's source object to graph
        set time threshold for E's source object to time of E
      }
      add edge from E's source object to E's sink object
    }
  }
}
```

(a) Algorithm for backward dependency graph

```
/* read events from earliest to latest */
foreach event E in log {
  foreach object O in graph {
    if(O is affected by E by the time thresh. for obj. O) {
      if(E's sink object not already in graph) {
        add E's sink object to graph
        set time threshold for E's sink object to time of E
      }
      add edge from E's source object to E's sink object
    }
  }
}
```

(b) Algorithm for forward dependency graph

Figure 3.1: Constructing a dependency graph

This code shows the basic algorithm used to construct (a) a backward dependency graph and (b) a forward dependency graph from a log of dependency-causing events with discrete times.

how to select the objects and events in the graph that relate to the attack.

Dependency graphs can represent the flow of information either backward in time or forward in time. In both cases, the graph starts from a single object, referred to as a *detection point*. Starting from a detection point, our goal is to build a dependency graph of all objects and events that causally affect [51] the state of the detection point (backward information flow) or are causally affected by the state of the detection point (forward information flow). The part of the our system that builds this dependency graph is called GraphGen. GraphGen is typically run off-line, i.e. after the attack.

3.3.1 Backward dependency graphs

To construct backward dependency graphs, GraphGen reads the logs of events, starting from the last event and reading toward the beginning of the log (Figure 3.1a). For each event, GraphGen evaluates whether that event can affect any object that is currently in the dependency graph. Each object in the evolving graph has a time threshold associated with it, which is the maximum time that an event can occur and be considered relevant for that object. GraphGen is initialized with the object associated with the detection point, and the time threshold associated with this object is the earliest time at which the administrator knows the object's state is compromised. Because the log is processed in reverse time order, all events encountered in the log after the detection point will occur before the time threshold of all objects currently in the graph.

Consider how this algorithm works for the set of events shown in Figure 3.2a (Figure

3.2b pictures the log of events as a complete dependency graph):

1. GraphGen is initialized with the detection point, which is file X at time 10. That is, the administrator knows that file X has the wrong contents by time 10.

2. GraphGen considers the event at time 8. This event does not affect any object in the current graph (i.e. file X), so we ignore it.

3. GraphGen considers the event at time 7. This event also does not affect any object in the current graph.

4. GraphGen considers the event at time 6. This event affects file X in time to affect its contents at the detection point, so GraphGen adds process C to the dependency graph with an edge from process C to file X. GraphGen sets process C's time threshold to be 6, because only events that occur before time 6 can affect C in time to affect the detection point.

5. GraphGen considers the event at time 5. This event affects an object in the dependency graph (process C) in time, so GraphGen adds file 1 to the graph with an edge to process C (at time 5).

6. GraphGen considers the event at time 4. This event affects an object in the dependency graph (process C) in time, so GraphGen adds process A to the dependency graph with an edge to process C (at time 4).

7. GraphGen considers the event at time 3. This event affects process A in time, so we add file 0 to the graph with an edge to process A (at time 3).

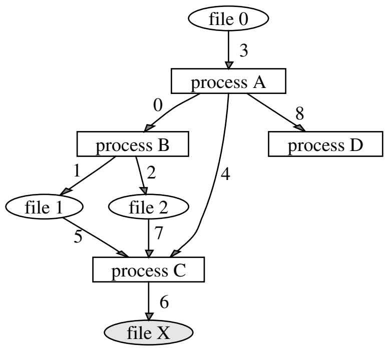
8. GraphGen considers the event at time 2. This event does not affect any object in the current graph.

9. GraphGen considers the event at time 1. This event affects file 1 in time, so we add process B to the graph with an edge to file 1 (at time 1).

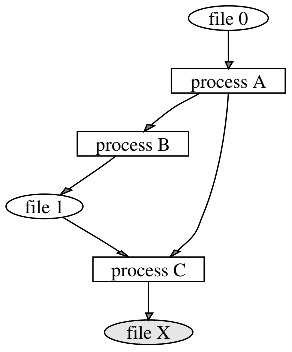
10. GraphGen considers the event at time 0. This event affects process B in time, so we add an edge from process A to process B (process A is already in the graph).

time 0: process A creates process B
time 1: process B writes file 1
time 2: process B writes file 2
time 3: process A reads file 0
time 4: process A creates process C
time 5: process C reads file 1
time 6: process C writes file X
time 7: process C reads file 2
time 8: process A creates process D

(a) event log



(b) dependency graph for complete event log



(c) dependency graph generated by GraphGen

Figure 3.2: Backward dependency graph for an example with discrete times

The label on each edge shows the time of the event. The detection point is file X at time 10. By processing the event log, GraphGen prunes away events and objects that do not affect file X by time 10.

The resulting backward dependency graph (Figure 3.2c) is a subset of the graph in Figure 3.2b. We believe this type of graph to be a useful picture of the events that lead to the detection point.

The full backward graph generation algorithm is a bit more complicated because it must handle events that span an interval of time, rather than events with discrete times. Consider a scenario where the dependency graph currently has an object O with time threshold t. If an event P⇒O occurs during time interval [x-y], then we should add P to the dependency graph iff $x < t$, i.e. this event started to affect O by O's time threshold. If P is added to the dependency graph, the time threshold associated with P would be minimum(t,y), because the event would have no relevant effect on O after time t, and the event itself stopped after time y.

Events with intervals are added to the log in order of the *later* time in their interval. This order guarantees that GraphGen sees the event and can add the source object for that event as soon as possible (so that the added source object can in turn be affected by events processed subsequently by GraphGen).

For example, consider how GraphGen would handle an event process B⇒file 1 in Figure 3.2b with a time interval of 1-7. GraphGen would encounter this event at a log time 7 because events are ordered by the later time in their interval. At this time, file 1 is not yet in the dependency graph. GraphGen remembers this event and continually re-evaluates whether it affects new objects as they are added to the dependency graph. When file 1 is added to the graph (log time 5), GraphGen sees that the event process B⇒file 1 affects file 1 and adds process B to the graph. The time threshold for process B would be time 5 (the lesser of time 5 and time 7).

GraphGen maintains several data structures to accelerate its processing of events. Its main data structure is a hash table of all objects currently in the dependency graph, called GraphObjects. GraphGen uses GraphObjects to determine quickly if the event under consideration affects an object that is already in the graph. GraphGen also remembers those events with time intervals that include the current time being processed in the log. GraphGen stores these events in an ObjectsIntervals hash table, hashed on the sink object for that event. When GraphGen adds an object to GraphObjects, it checks if any events in the ObjectsIntervals hash table affect the new object before the time threshold for the new object. Finally, GraphGen maintains a priority queue of events with intervals that include the current time (prioritized by the starting time of the event). The priority queue allows GraphGen to find and discard events quickly whose intervals no longer include the current time.

3.3.2 Forward dependency graphs

Backward dependency graphs show which events preceding an intrusion detection point could have contributed to the modified state or event that was detected. We would like to generalize this approach to analyze in the forward direction. Analyzing causality in the backward direction answers the question "How did these events and state get on my system?" whereas analyzing causality in the forward direction answers the question "What events and state were affected by the intrusion detection point?". Consider a scenario in which an attacker replaces the /bin/ls executable with a program that sends a user's private files to a collection site. Once one detects that /bin/ls was modified, forward tracking will show which files of which users were leaked as a result; this could help limit the damage done by the intrusion (e.g., by informing the user which credit cards should be canceled).

Forward causal analysis is a straightforward extension to backward analysis. Backward information-flow graphs add events to a graph if they affect an object before that object's time threshold, forward tracking adds events to a graph if they are affected by an object after that object's time threshold (Figure 3.1).

3.4 Dependencies tracked by current prototype

Section 3.2 lists numerous ways in which one object can potentially affect another. It is important to note, however, that *affecting* an object is not the same as *controlling* an object. Dependency-causing events vary widely in terms of how much the source object can control the sink object. Our current implementation focuses on tracking the events we consider easiest for an attacker to use to accomplish a task; we call these events *high-control events*.

Some examples of high-control events are changing the contents of a file or creating a child process. It is relatively easy for an intruder to perform a task by using high-control events. For example, an intruder can install a backdoor easily by modifying an executable file, then creating a process that executes it.

Some examples of low-control events are changing a file's access time, creating a filename in a directory, or enumerating a registry key. Although these events can affect the execution of other processes, they tend to generate a high degree of noise in the dependency graph. For example, if EventLogger tracks the dependency caused by reading a directory, then a process that lists the files in /tmp would depend on all processes that have ever created, renamed, or deleted filenames in /tmp. Timing channels [52] are an example of an extremely low-control event; e.g., an attacker may be able to trigger a race condition by executing a CPU-intensive program.

Fortunately, we are able to provide useful information without tracking low-control events, even if low-control events are used in the attack. This is because it is difficult for an intruder to perform a task solely by using low-control events. Consider an intruder who wants to use low-control events to accomplish an arbitrary task; for example, he may try to cause a program to install a backdoor when it sees a new filename appear in /tmp.

Using an existing program to carry out this task is difficult because existing programs do not generally perform arbitrary tasks when they see incidental changes such as a new filename in /tmp. If an attacker can cause an existing program to perform an arbitrary task by making such an incidental change, it generally means that the program has a bug (e.g., buffer overflow or race condition). Even if EventLogger does not track this event, GraphGen will still be able to highlight the buggy existing program by tracking the chain of events from the detection point back to that program.

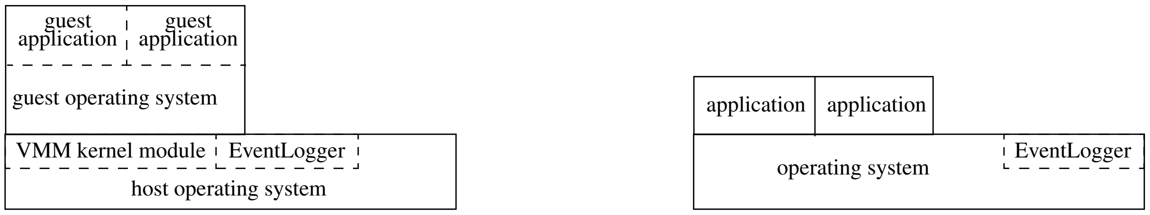
Using a new, custom program to carry out an arbitrary task is easy. However, it will not evade GraphGen's information flow graph because the events of writing and executing such a custom program are high-control events and GraphGen will link the backdoor to the intruder's earlier actions through those high-control events. To illustrate this, consider in Figure 3.2b if the event "file 1⇒process C" was a low-control event, and process C was created by process B (rather than by process A as shown). Even if EventLogger did not track the event "file 1⇒process C", it would still link process B to the detection point via the event "process B⇒process C".

EventLogger currently logs the following high-control events: process creation through fork or clone; load and store to shared memory; read and write of files and pipes; sending and receiving data from a socket; execve of files; load and store to mmap'ed files; and setting and querying registry values. We have implemented the logging and tracking of file attributes and filename create, delete, and rename (these events are not reflected in Section 4.2's results).

3.5 Implementation structure of EventLogger and GraphGen

There are several ways to implement EventLogger, and the resulting event logs are the same independent of where EventLogger is implemented.

One strategy for our prototype is to run the target operating system (Linux 2.4.18) and applications inside a virtual machine and to have the virtual-machine monitor call an EventLogger procedure at appropriate times (Figure 3.3). The operating system running inside the virtual machine is called the *guest operating system* to distinguish it from the



(a) virtual-machine implementation

(b) standalone implementation

Figure 3.3: System structures for logging events

We have implemented EventLogger in two ways. In the virtual-machine implementation (Figure 3.3a), we run the target operating system and applications in a virtual machine and log events in the virtual-machine monitor running below the target operating system. In response to causal events, the virtual-machine monitor (VMM) module calls an EventLogger kernel procedure, then EventLogger reads information about the event from the virtual machine's physical memory. In the standalone implementation (Figure 3.3b), we run applications directly on the host operating system and log events from within that operating system.

operating system that the virtual machine is running on, which is called the *host operating system*. Guest processes run on the guest operating system inside the virtual machines; host processes run on the host operating system. The entire virtual machine is encapsulated in a host process. The log written by EventLogger is stored on the host. The virtual-machine monitor prevents intruders in the guest from interfering with EventLogger or its log.

EventLogger gleans information about events and objects inside the target system by examining the state of the virtual machine. The virtual-machine monitor notifies EventLogger whenever a guest application invokes or returns from a system call or when a guest application process exits. EventLogger learns about the event from data passed by the virtual-machine monitor and from the virtual machine's physical memory (which is a host file). EventLogger is compiled with headers from the guest kernel and reads guest kernel data structures from the guest's physical memory to determine event information (e.g., system call parameters), object identities (e.g., file inode numbers, filenames, process identifiers) and dependency information (e.g., it reads the address map of a guest process to learn what mmap'ed files it inherited from its parent). The code for EventLogger is approximately 1300 lines, and we added 40 lines of code to the virtual-machine monitor to support EventLogger. We made no changes to the guest operating system.

One of the standard reasons for using a virtual machine—correctness in the presence of a compromised target operating system—does not hold for EventLogger. If an attacker gains control of the guest operating system, she can carry out arbitrary tasks inside the guest without being tracked by EventLogger.

We use a version of the UMLinux virtual machine [15] that uses a host kernel (based on Linux 2.4.18) that is optimized to support virtual machines [46]. The virtualization overhead of the optimized UMLinux is comparable to that of VMWare Workstation 3.1. CPU-intensive applications experience almost no overhead, and kernel-intensive applications such as SPECweb99 and compiling the Linux kernel experience 14-35% overhead [46].

A second strategy we use is to add EventLogger directly to the target operating system and not use a virtual machine. To protect EventLogger's log from the intruder, we store the log on a remote computer or in a protected file on the local computer. We have ported EventLogger to a standalone operating system (Linux 2.4.20) and to Windows 2000/XP to have the option of using our system without using a virtual machine. To port EventLogger to the target operating system, we modified only the code that gleans information about events and objects.

To track information-flow across multiple hosts, EventLogger includes network sends and receives. Connecting a send event with its corresponding receive event requires identifying each packet (or connection). There are several ways to identify packets. The simplest is to use information in existing network headers, such as the source and destination IP address, port number, and sequence number (for TCP messages)[1]. Another approach is to supplement messages with additional information [60], either at network routers or using our modified kernel. Finally, one could identify packets by storing a hash of their contents. We currently identify packets by their source and destination addresses and sequence number; this simple approach is sufficient for the TCP-based attacks evaluated in this dissertation.

[1]The gateway that receives messages from outside the network can use ingress filtering to check that these messages do not spoof an internal network address [36]

EventLogger logs can be stored using either the filesystem or a database. Storing logs in a file is convenient, but using a database allows for quicker graph generation. The key observation is that only a small subset of the total objects in a log end up in an information flow graph. Using a file, GraphGen must process all events sequentially, even if the objects are not present in the graph. Using a database, only events for objects of interest are queried, so graph generation time scales with the number of object in the graph, not the total number of objects in the system. This allows for efficient graph generation, even when large logs must be processed.

GraphGen uses EventLogger logs to generate information flow graphs. GraphGen produces a graph in a format suitable for input to the dot program (part of AT&T's Graph Visualization Project), which generates the human-readable graphs used in this paper. GraphGen runs on any computer used by the administrator and does not require any data from the computer in question other than the EventLogger log.

CHAPTER 4

Analyzing Intrusions On a Single Host

In this chapter we discuss how information-flow graphs can be used to help analyze intrusions on a single host. We first talk about how GraphGen prioritizes portions of information-flow graphs to highlight a subset of objects and events. Next, we describe a series of attacks that we analyze using BackTracker to determine the application that was originally exploited. Then, we discuss advantages and disadvantages of ForwardTracker and present an illustrative example.

4.1 Prioritizing portions of an information-flow graph

Dependency graphs for a busy system may be too large to scrutinize each object and event. Fortunately, not all objects and events warrant the same amount of scrutiny when a system administrator analyzes an intrusion. This section describes several ways to prioritize or filter a dependency graph in order to highlight those parts that are mostly likely to be helpful in understanding an intrusion. Of course, there is a tradeoff inherent to any filtering. Even objects or events that are unlikely to be important in understanding an intrusion may nevertheless be relevant, and these false negatives may accidentally hide important sequences of events.

One way to prioritize important parts of a graph is to ignore certain objects. For example, the login program reads and writes the file /var/run/utmp. These events cause a new login session to depend on all prior login sessions. Another example is the file /etc/mtab. This file is written by mount and umount and is read by bash at startup, causing all events to depend on mount and umount. A final example is that the bash shell commonly writes to a file named .bash_history when it exits. Shell invocations start by reading .bash_history, so all actions by all shells depend on all prior executions of bash. While these are true dependencies, it is easier to start analyzing the intrusion without these objects cluttering the graph, then to add these objects if needed.

A second way to prioritize important parts of a graph is to filter out certain types of events. For example, one could filter out some low-control events.

These first two types of filtering (objects and events) may filter out a vital link in the intrusion and thereby disconnect the detection point from the source of the intrusion. Hence they should be used only for cases where they reduce noise drastically with only a small risk of filtering out vital links. The remainder of the filtering rules do not run the risk of breaking a vital link in the middle of an attack sequence.

A third way to simplify the graph is to hide files that have been read but not written in the time period being analyzed (read-only files). For example, in Figure 3.2c, file 0 is read by process A but is not written during the period being analyzed. These files are often default configuration or header files. Not showing these files in the graph does not generally hinder one's ability to understand an attack because the attacker did not modify these files in the time period being considered and because the processes that read the files are still included in the dependency graph. If the initial analysis does not reveal enough about the attack, an administrator may need to extend the analysis further back in the log to include events that modified files which were previously considered read-only. Filtering out read-only files cannot break a link in any attack sequence contained in the log being analyzed, because there are no events in that log that affect these files.

A fourth way to prioritize important parts of a graph is to filter out helper processes that take input from one process, perform a simple function on that input, then return data to the main process. For example, the system-wide bash startup script (/etc/bashrc) causes bash to invoke the id program to learn the name and group of the user, and the system startup scripts on Linux invoke the program consoletype to learn the type of the console that is being used. These usage patterns are recognized easily in a graph: they form a cycle in the graph (usually connected by a pipe) and take input only from the parent process and from read-only files. As with the prior filtering rule, this rule cannot disconnect a detection point from an intrusion source that precedes the cycle, because these cycles take input only from the main process, and the main process is left in the dependency graph.

A fifth way to prioritize important parts of a graph is to choose *several* detection points, then take the intersection of the dependency graphs formed from those dependency points. The intersection of the graphs is likely to highlight the earlier portion of an attack (which affect all detection points), and these portions are important to understanding how the attacker initially gained control in the system.

We implement these filtering rules as options in GraphGen. GraphGen includes a set of default rules which work well for all attacks we have experienced. A user can add to a configuration file regular expressions that specify additional objects and events to filter. We considered filtering the graph after GraphGen produced it, but this would leave in objects that should have been pruned (such as an object that was connected only via an object that was filtered out).

Other graph visualization techniques can help an administrator understand large dependency graphs. For example, a post-processing tool can aggregate related objects in the graph, such as all files in a directory, or show how the graph grows as the run progresses.

We expect an administrator to run GraphGen several times with different filtering rules and log periods. He or she might first analyze a short log that he or she hopes includes the entire attack. He or she might also filter out many objects and events to try to highlight the most important parts of an intrusion without much noise from irrelevant events. If this initial analysis does not reveal enough about the attack, he or she can extend the analysis period further back in the log and use fewer filtering rules.

4.2 BackTracking attacks

This section evaluates how well BackTracker works on three real attacks and one simulated attack (Table 4.1).

To experience and analyze real attacks, we set up a honeypot machine [20] [63] and installed the default configuration of RedHat 7.0. This configuration is vulnerable to several remote and local attacks, although the virtual machine disrupts some attacks by shrinking the virtual address space of guest applications. Our honeypot configuration is vulnerable

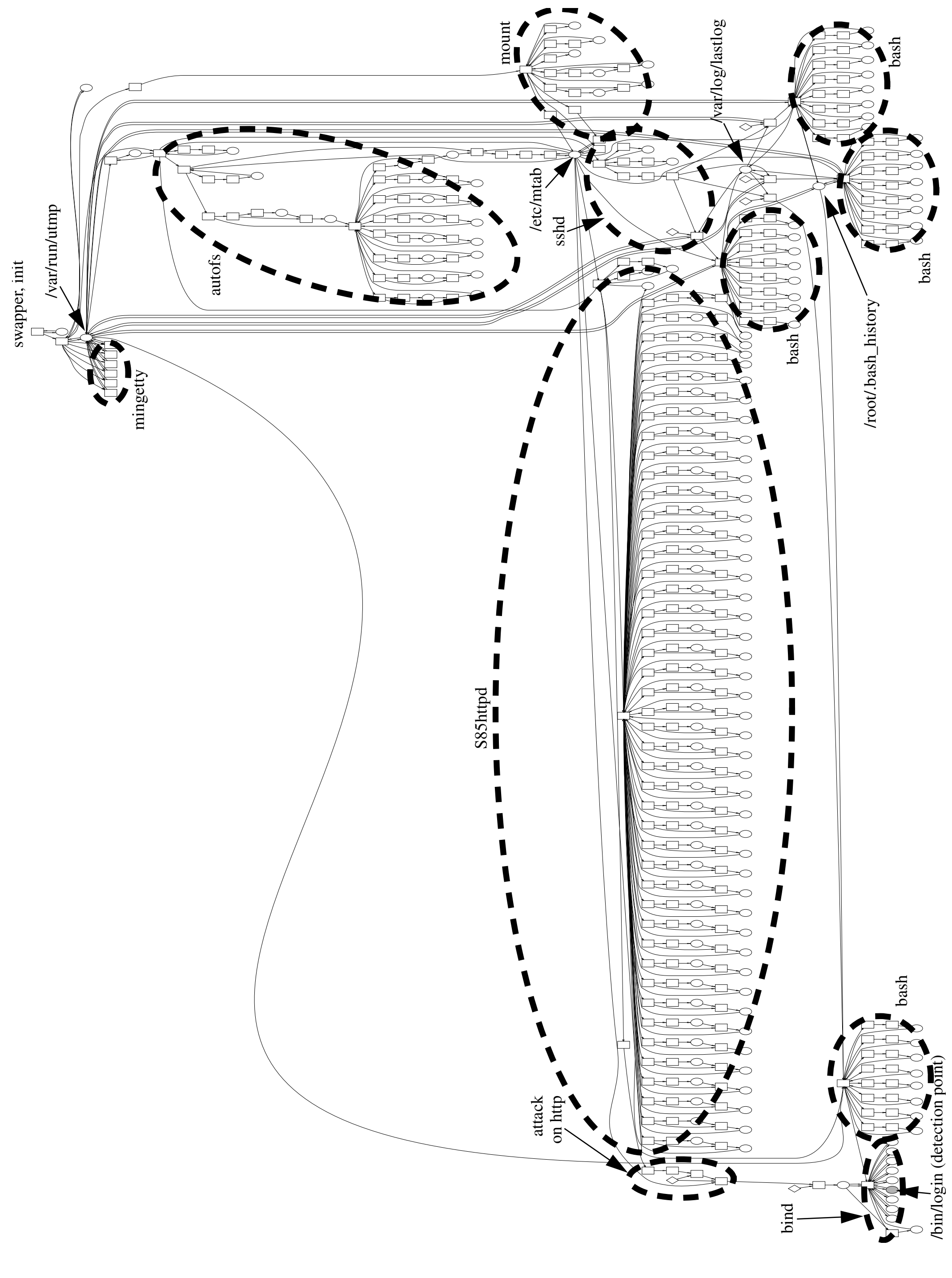


Figure 4.1: Mostly-unfiltered dependency graph for *bind attack*

The only filtering used was to remove files that were read but not written. The circled areas and labels identify the major portions of the graph. Of particular interest are the files we filter out in later dependency graphs: /var/run/utmp, /etc/mtab, /var/log/lastlog, /root/.bash_history. We will also filter out helper processes that take input from one process (usually via a pipe), perform a simple function on that input, then return data to the main process. Most objects associated with S85httpd are helper processes spawned by find when S85httpd starts.

	***bind* (Fig 4.1-4.2)**	***ptrace* (Fig 4.3)**	***openssl-too* (Fig 4.4)**	***self* (Fig 4.5)**
time period being analyzed	24 hours		61 hours	24 hours
# of objects and events in log	155,344 objects 1,204,166 events		77,334 objects 382,955 events	2,187,963 objects 55,894,869 events
# of object and events in unfiltered dependency graph	5,281 objects 9,825 events	552 objects 2,635 events	495 objects 2,414 events	717 objects 3,387 events
# of objects and events in filtered dependency graph	24 objects 28 events	20 objects 25 events	28 objects 41 events	56 (36) objects 81 (49) events
growth rate of EventLogger's log	0.017 GB/day		0.002 GB/day	1.2 GB/day
time overhead of EventLogger	0%		0%	9%

Table 4.1: Statistics for BackTracker's analysis of attacks

This table shows results for three real attacks and one simulated attack. Event counts include only the first event from a source object to a sink object. GraphGen and the filtering rules drastically reduce the amount of information that an administrator must peruse to understand an attack. Results related to EventLogger's log are combined for the bind and ptrace attacks because these attacks are intermingled in one log. Object and events counts for the *self attack* are given for two different levels of filtering.

---

to (at least) two attacks. A remote user can exploit the OpenSSL library used in the Apache web server (httpd) to attain a non-root shell [18], and a local user can exploit sendmail to attain a root shell [24]. After an attacker compromises the system, they have more-or-less free reign on the honeypot—they can read files, download, compile, and execute programs, scan other machines, etc.

We ran a variety of tools to detect intruders. We used a home-grown imitation of TripWire [44] to detect changes to important system files. We used Ethereal and Snort to detect suspicious amounts of incoming or outgoing network traffic. We also perused the system manually to look for any unexpected files or processes.

We first evaluate how necessary it is to use the filtering rules described in Section 4.1. Consider an attack we experienced on March 12, 2003 that we named the *bind attack*. The machine on this day was quite busy: we were the target of two separate attacks (the *bind attack* and the *ptrace attack*), and one of the authors logged in several times to use the machine (mostly to look for signs of intruders, e.g., by running netstat, ps, ls, pstree). We detected the attack by noticing a modified system binary (/bin/login). EventLogger's log for this analysis period covered 24 hours and contained 155,344 objects and 1,204,166 events (all event counts in this paper count only the first event from a specific source object to a specific sink object).

Without any filtering, the dependency graph generated by GraphGen for this attack contains 5,281 objects and 9,825 events. While this is two orders of magnitude smaller than the complete log, it is still far too many events and objects for an administrator to analyze easily. We therefore consider what filtering rules we can use to reduce the amount of information presented to the administrator, while minimizing the risk of hiding important steps in the attack.

Figure 4.1 shows the dependency graph generated by GraphGen for this attack after filtering out files that were read but not written. The resulting graph contains 575 objects and 1,014 events. Important parts of the graph are circled or labeled to point out the filtering rules we discuss next.

Significant noise comes from several root login sessions by one of the authors during the attack. The author's actions are linked to the attacker's actions through /root/.bash_-history, /var/log/lastlog, and /var/run/utmp. /etc/mtab also generates a lot of noise, as it is written after most system startup scripts and read by each bash shell. Finally, a lot of noise is generated by helper processes that take input only from their parent process, perform a simple function on that input, then return data to the parent (usually through a pipe). Most processes associated with S85httpd on the graph are helper processes spawned by find when S85httpd starts.

Figure 4.2 shows the dependency graph for the *bind attack* after GraphGen applies the following filtering rules: ignore files that were read but not written; ignore files /root/.bash_history, /var/run/lastlog, /var/run/utmp, /etc/mtab; ignore helper processes that take input only from their parent process and return a result through a pipe. We use these same filtering rules to generate dependency graphs for all attacks.

These filtering rules reduce the size of the graph to 24 objects and 28 events, and

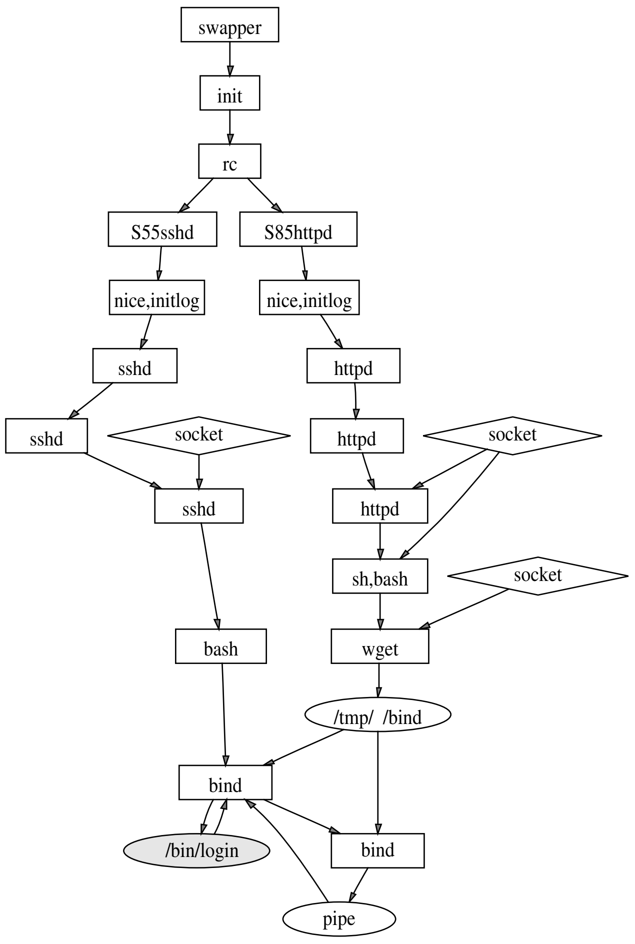


Figure 4.2: Filtered dependency graph for *bind attack*.

Processes are shown as boxes (labeled by program names called by execve during that process's lifetime); files are shown as ovals; sockets are shown as diamonds. BackTracker can also show process IDs, file inode numbers, and socket ports. The detection point is shaded.

---

make the *bind attack* fairly easy to analyze. The attacker gained access through httpd, downloaded a rootkit using wget, then wrote the rootkit to the file "/tmp/ /bind". Sometime later, one of the authors logged in to the machine, noticed the suspicious file, and decided to execute it out of curiosity (don't try this at home!). The resulting process installed a number of modified system binaries, including /bin/login. This graph shows that BackTracker can track across several login sessions. If the attacker had installed /bin/login without being noticed, then logged in later, we would be able to backtrack from a detection point in her second session to the first session by her use of the modified /bin/login.

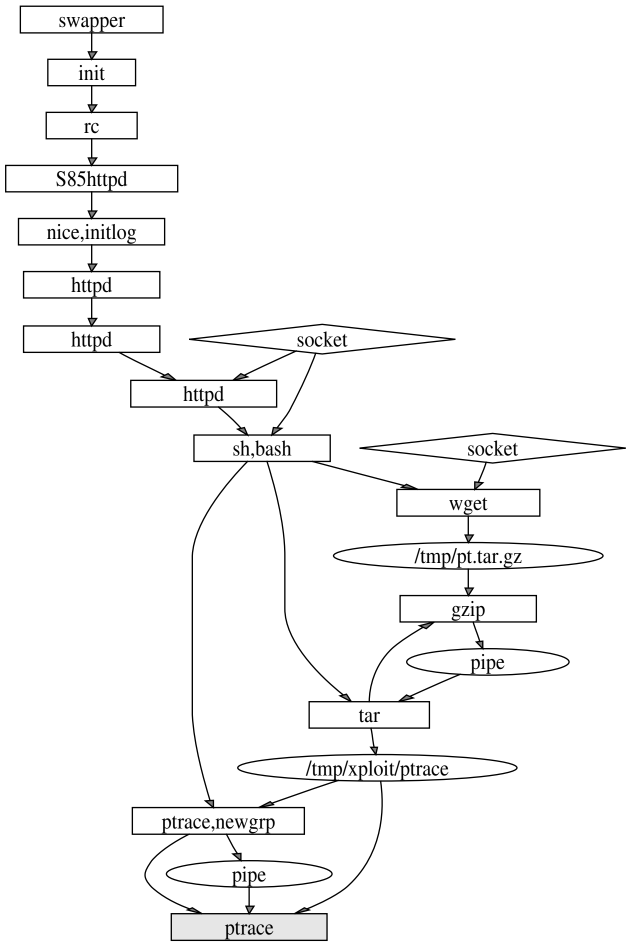


Figure 4.3: Filtered dependency graph for *ptrace* attack

Figure 4.3 shows the filtered dependency graph for a second attack that occurred in the same March 12, 2003 log, which we named the *ptrace attack*. The intruder gained access through httpd, downloaded a tar archive using wget, then unpacked the archive via tar and gzip. The intruder then executed the ptrace program using a different group identity. We later detected the intrusion by seeing the ptrace process in the process listing. We believe the ptrace process was seeking to exploit a race condition in the Linux ptrace code to gain root access. Figures 4.3 and 4.2 demonstrate BackTracker's ability to separate two intermingled attacks from a single log. Changing detection points from /bin/login to ptrace is sufficient to generate distinct dependency graphs for each attack.

Figure 4.4 shows the filtered dependency graph for an attack on March 2, 2003 which we named the *openssl-too attack*. The machine was used lightly by one of the authors

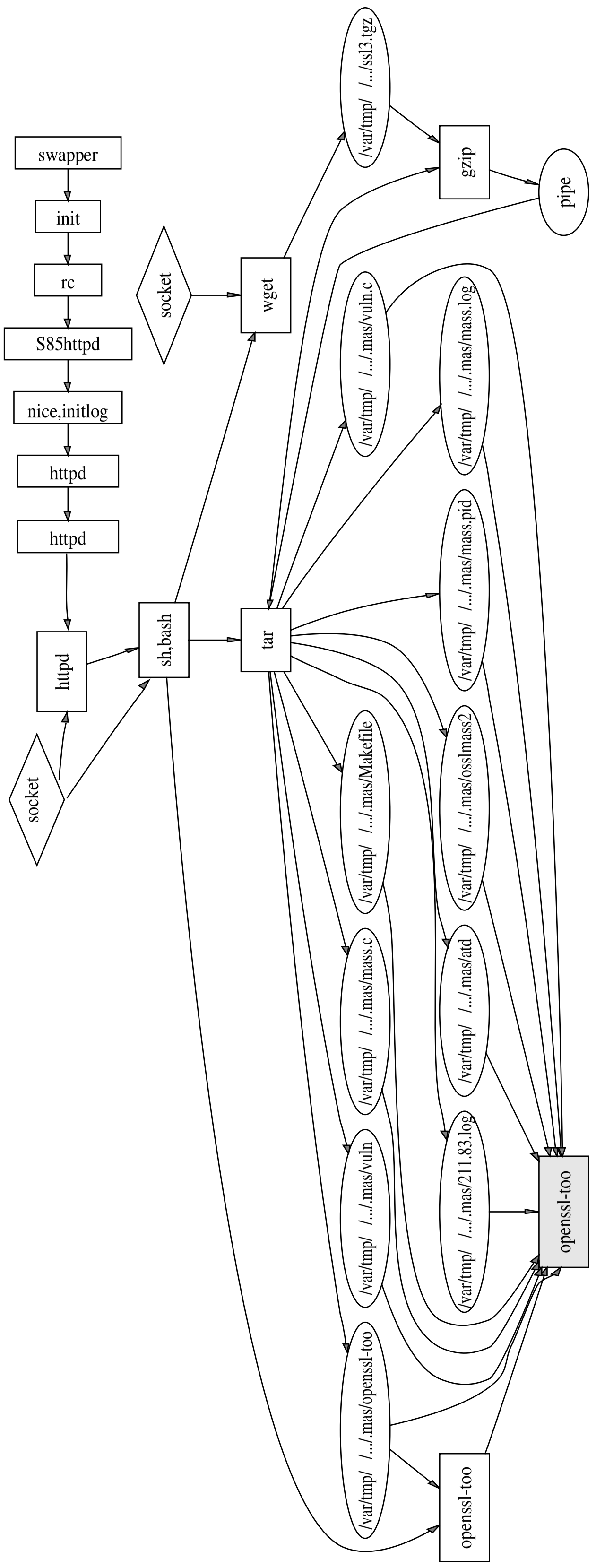

swapper
init
rc
S85httpd
nice,initlog
httpd
httpd
httpd
socket
sh,bash
socket
wget
tar
/var/tmp/ /.../ssl3.tgz
gzip
pipe
/var/tmp/ /.../.mas/vuln.c
/var/tmp/ /.../.mas/mass.log
/var/tmp/ /.../.mas/mass.pid
/var/tmp/ /.../.mas/osslmass2
/var/tmp/ /.../.mas/atd
/var/tmp/ /.../.mas/211.83.log
/var/tmp/ /.../.mas/Makefile
/var/tmp/ /.../.mas/mass.c
/var/tmp/ /.../.mas/vuln
/var/tmp/ /.../.mas/openssl-too
openssl-too
openssl-too

Figure 4.4: Filtered dependency graph for *openssl-too attack*.

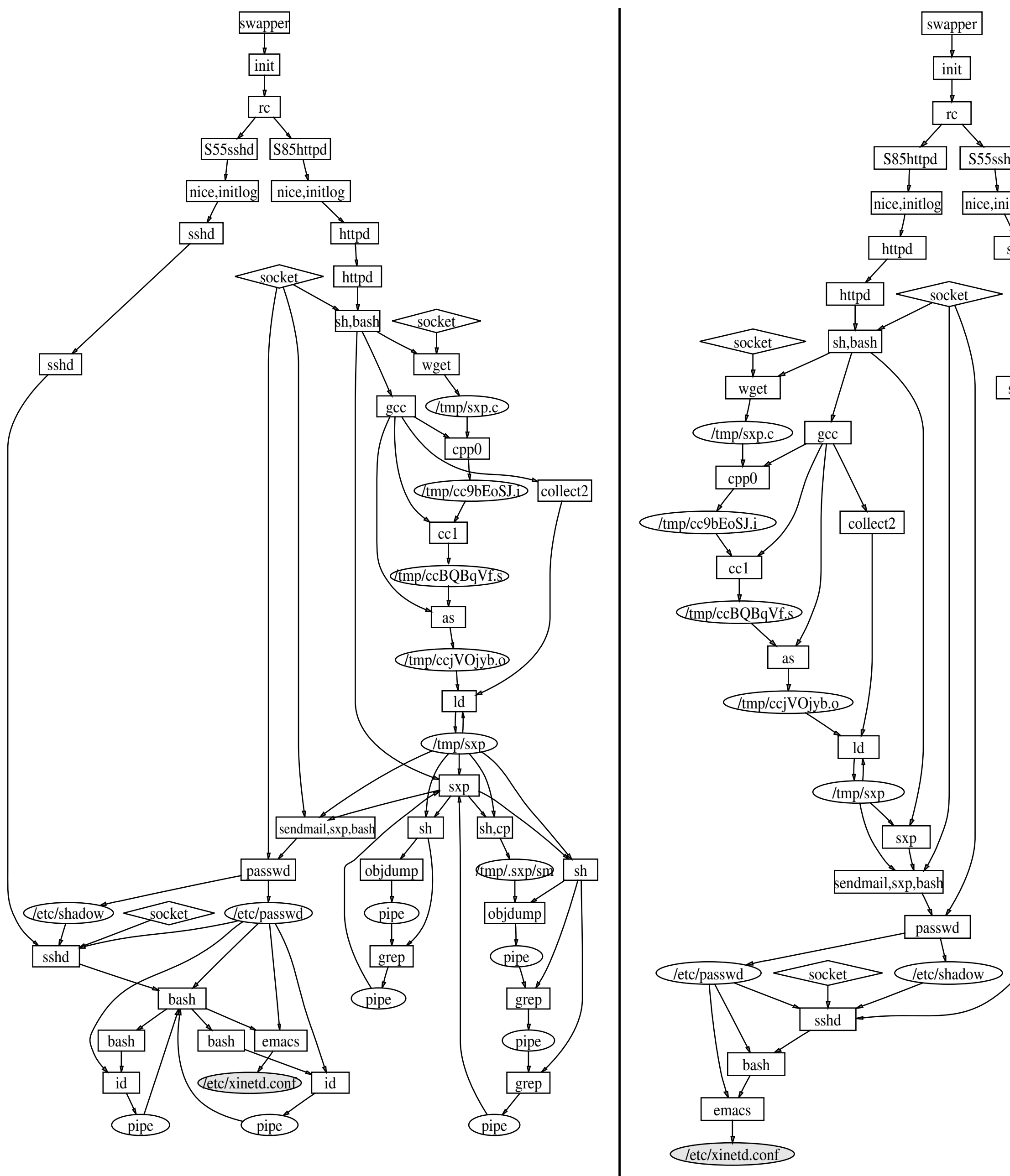
(to check for intrusions) during the March 1-3 period covered by this log. The attacker gained access through httpd, downloaded a tar archive using wget, then installed a set of files using tar and gzip. The attacker then ran the program openssl-too, which read the configuration files that were unpacked. We detected the intrusion when the openssl-too process began scanning other machines on our network for vulnerable ports.

Another intrusion occurred on our machine on March 13, 2003. The filtered dependency graph for this attack is almost identical to the *ptrace attack*.

Figure 4.5a shows the default filtered dependency graph for an attack we conducted against our own system (*self attack*). *Self attack* is more complicated than the real attacks we have been subjected to. We gain unprivileged access via httpd, then download and compile a program (sxp) that takes advantage of a local exploit against sendmail. When sxp runs, it uses objdump to find important addresses in the sendmail binary, then executes sendmail through execve to overflow an argument buffer and provide a root shell. We use this root shell to add a privileged user to the password files. Later, we log into the machine using this new user and modify /etc/xinetd.conf. The detection point for this attack is the modified /etc/xinetd.conf.

One goal for this attack is to load the machine heavily to see if BackTracker can separate the attack events from normal events. Over the duration of the workload, we continually ran the SPECweb99 benchmark to model the workload of a web server. To stress the machine further, we downloaded, unpacked, and continually compiled the Linux kernel. We also logged in several times as root and read /etc/xinetd.conf. The dependency graph shows that BackTracker separates this legitimate activity from the attack.

We anticipate that administrators will run GraphGen multiple times with different filtering rules to analyze an attack. An administrator can filter out new objects and events easily by editing the configuration file from which GraphGen reads its filter rules. Figure 4.5b shows the dependency graph generated with an additional rule that filters out all pipes. While this rule may filter out some portions of the attack, it will not usually disconnect the detection point from the from an intrusion source, because pipes are inherited from a process's ancestor, and BackTracker will track back to the ancestor through process creation events. In Figure 4.5, filtering out pipes eliminates objdump, which is related to



(a) default filtering rules

(b) default filtering rules + filter out pipes

Figure 4.5: Filtered dependency graph for *self attack*

Figure 4.5a shows the dependency produced by GraphGen with the same filtering rules used to generate Figures 4.3, 4.2, and 4.4. Figure 4.5b shows the dependency graph produced by GraphGen after adding a rule that filters out pipes. Figure 4.5b is a subgraph of Figure 4.5a.

the attack but not critical to understanding the attack.

Next, we measure the space and time overhead of EventLogger (Table 4.1). For the real attacks, the system is idle for long periods of time. The average time and space overhead for EventLogger is very low for these runs because EventLogger only incurs overhead when applications are actively using the system.

The results for *self attack* represent what the time and space overheads would be like for a system that is extremely busy. In particular, serving web pages and compiling the Linux kernel each invoke a huge number of relevant system calls. For this run, EventLogger slows the system by 9%, and its compressed log grows at a rate of 1.2 GB/day. While this is a substantial amount of data, a modern hard disk is large enough to store this volume of log traffic for several months.

GraphGen is run after the attack (off-line), so its performance is not as critical as that of EventLogger. On a 2.8 GHz Pentium 4 with 1 GB of memory, GraphGen took about 26 seconds to process the largest log: the *self attack*. For this test, the log was stored in a MySQL database. Each of the real attacks took less time to generate the graph.

4.3 ForwardTracking attacks

This section discusses ForwardTracker and why tracking attacks forward is fundamentally more difficult than tracking them backward. We then describe a Windows XP experiment we use to evaluate ForwardTracker, as well as the Windows specific filtering techniques we use. Finally, we show how our example illustrates both what is useful about the ForwardTracking technique, as well as some of the difficulties.

Tracking attacks forward is a fundamentally more difficult problem then tracking attacks backward. The main difference is the question being asked. When BackTracking an attack, we try to answer the question “how did the attacker gain access to my system?”. BackTracker aims to highlight the single application which lead to the attack in question. Filtering events and objects is acceptable as long as the application that was exploited remains in the information flow graph. When ForwardTracking an attack, we try to answer a different question: “what did the attacker do after they broke in?”. If the administrator filters *anything* from the graph, they remove part of the attacker’s actions after they broke in. Thus, ForwardTracking forces administrators to tradeoff excessive tainting vs completeness. In addition to answering a more general question, the ForwardTracking problem is further complicated by the process model used by modern operating systems. Specifically, processes, by definition, have exactly one parent process and can have an unbounded number of children processes. When BackTracking an attack, GraphGen identifies the one parent process, but going forward, GraphGen identifies all children processes. This conservative approach to handling causal events can lead to false positives and innocent objects implicated in an attack.

To illustrate the benefits and shortcomings of ForwardTracker, we analyze a real attack on a honeypot system running an unpatched version of Windows XP. Two most commonly exploited vulnerabilities we encountered were the RPC daemon vulnerability [3] and the local security authentication server service (lsass) vulnerability [5].

Tracking attacks for Windows XP requires additional filtering rules to cope with the Windows registry. Windows uses the registry as a central repository for application and system wide configuration data; most of this data is application-specific configuration data that does not give attackers high levels of control over the system. For example, a word processor might use the registry to store a list of the last few active documents, or to store the last size and location of your editor's window. However, some well-known registry entries do provide attackers with a high level of control over the system. These registry entries are called Auto-Start Extensibility Points (ASEPs) and are discussed in detail in [72]. Hackers leverage these ASEPs to load device drivers into the system, install services that automatically restart upon reboot, and force malicious dynamically loaded libraries (dlls) into the address space of innocent applications. As a result, we track all registry events involving ASEPs.

Figure 4.6 shows the backward dependency graph for an attack observed on April 20, 2005 that we call the *mslaugh* attack. We detected this attack by manually searching through our EventLogger log to find anomalous process creation events. We found the process creation event for a process called mslaugh.exe, which is a known variant of the blaster worm [4]. Using the mslaugh.exe process as the starting point for our backward analysis, we determined that the svchost process was compromised since it does not typ-

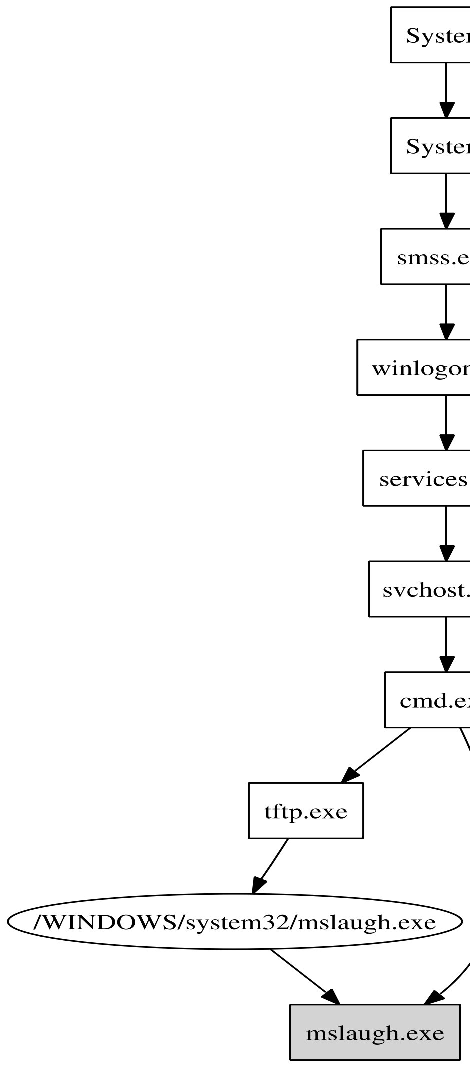


Figure 4.6: Backward information-flow graph for *mslaugh* attack

ically download files using tftp or launch command shells. Thus, the svchost.exe process became the starting point for our forward analysis.

Figure 4.7 shows the forward dependency graph for the mslaugh attack starting from svchost.exe process. This graph illustrates both what is good about ForwardTracker, as well as potential shortcomings. One benefit of forward information-flow graphs is that we learn more about the attack. From Figure 4.7, we see that the attacker launches two mslaugh processes, not one, and both processes write ASEP registry values to force the mslaugh process to restart when the system reboots. This additional information was unavailable in the backward information-flow graph. However, the remainder of the graph

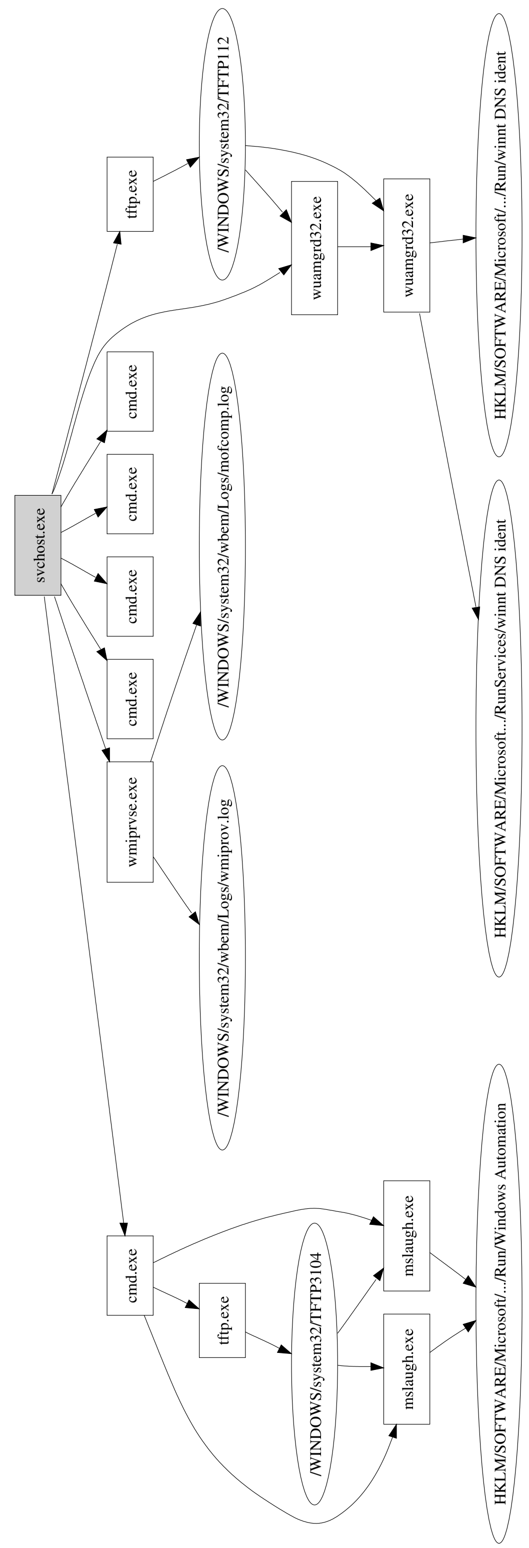

svchost.exe
cmd.exe
wmiprvse.exe
cmd.exe
cmd.exe
cmd.exe
cmd.exe
tftp.exe
tftp.exe
/WINDOWS/system32/wbem/Logs/wmiprov.log
/WINDOWS/system32/wbem/Logs/mofcomp.log
/WINDOWS/system32/TFTP112
/WINDOWS/system32/TFTP3104
wuamgrd32.exe
mslaugh.exe
mslaugh.exe
wuamgrd32.exe
HKLM/SOFTWARE/Microsoft/.../Run/Windows Automation
HKLM/SOFTWARE/Microsoft.../RunServices/winnt DNS ident
HKLM/SOFTWARE/Microsoft/.../Run/winnt DNS ident

Figure 4.7: Forward information-flow graph for *mslaugh* attack

shows potential shortcomings of the ForwardTracker technique. First, the wmiprvse.exe is a false positive. Wmiprvse.exe handles instrumentation management on Windows systems and is a legitimate service that happened to be started by the same svchost process as the RPC service. Second, there is also a wuamgrd.exe process, which is a a bot called agobot [6], that an attacker installed in a separate attack on the same vulnerable svchost process. The backward information-flow graph focuses in on the attack because, when going backward, we identify the single parent process. However, going forward we pick up all children, which is where some of this extra data came from. In addition to extra children, we also filter registry activity that is, strictly speaking, the result of the attack.

CHAPTER 5

Analyzing Intrusions Across Multiple Hosts

BackTracker and ForwardTracker's analysis is limited to events that occur on the same host as the detection point. Because many attacks propagate via the network, we would like to generalize this approach to track intrusions as they infect multiple hosts. Tracking intrusions across multiple hosts allow us to find other compromised hosts upstream (using backward causality) and downstream (using forward causality). As with host-level information flow, this type of tracking is limited to machines under our administrative control.

In this chapter, we describe how to follow attacks across the network using Bidirectional Distributed BackTracker (BDB) to track network send and receive events. We first introduce new filtering rules to prioritize the network level causal paths most likely to describe an intrusion as it traverses across multiple hosts. Next, we describe how to track multi-hop attacks. Finally, we discuss correlating multiple intrusion-detection alerts using information-flow graphs.

5.1 Prioritizing network connections

Tracking the causal relationships resulting from network communication can lead to extremely large graphs. Consider the following scenario: an intruder attains a login session on an internal computer (A). He or she then uses that login session to compromise an internal SMB server (B). From the SMB server, he or she browses the departmental web server, then launches a worm. Using one of the outgoing worm messages as the detection point, an administrator can generate a multi-host, backward causality graph. This graph will include the key link in the attack, which is the message from the login session (A) to the SMB server (B). However, the graph will also include the irrelevant messages from the departmental web server. To make matters worse, the graph may also include other clients that happened to affect the execution of the departmental web server before that server sent pages to the SMB server.

Another example scenario is if a network service handles a series of requests in a single process. If this network service is compromised, it depends on all prior incoming requests. Fortunately, network services are often built to create a new process to handle each (or a few) incoming requests. Creating a new process helps to limit the set of incoming packets that causally precede an intrusion.

In order to counteract the tendency of graphs to explode in size, we must prioritize which packets to include in multi-host causality graphs. Prioritizing packets leads to the same tradeoffs inherent to any kind of filtering. Even objects or events that are unlikely to be important in understanding an intrusion may nevertheless be relevant, and filtering these out may accidentally hide important sequences of events.

There are numerous methods one could use to prioritize which packets to follow in backward or forward causal analysis. The first method is a simple heuristic that works well for today's simple worms. A common pattern of today's worms is to connect to a network service, compromise it (e.g., with a buffer overflow), then immediately start a root shell or a backdoor process. For this pattern, the best process to follow when performing backward causality process is the highest (i.e., earliest) process in the backward graph that received a network packet. The best packet to follow for this pattern is the most recent packet received by the highest process. We call this heuristic *highest process, most recent packet*.

A more general and robust method for prioritizing packets is to choose packets that are related causally to another IDS alert. Some network messages are likely to be innocent or ineffective, while others may be malicious and effective. Messages that are malicious and effective are more likely to lead to other IDS alerts, thus any network packets connected to IDS alerts should be prioritized. Suspicious network packets may be connected directly to an alert from a network anomaly detector that highlights suspicious network messages, or they may be connected indirectly to an alert from a host IDS that detects suspicious host activity that is caused by network messages.

Another method of prioritizing packets leverages the fact that most worms repeat their actions as they traverse hosts. Worms usually propagate from host to host using the same exploit, or perhaps using one of a few exploits. They may also perform the same activities on each host they compromise, such as installing backdoors, patching vulnerabilities, or scanning for private information. These repeated actions form a pattern that characterizes the worm. As we follow an intrusion from host to host (forward or backward), we can learn this pattern and use it to prioritize packets that cause similar patterns on other hosts.

A pattern for a worm may be characterized in many ways. It could be the set of files or processes that causally follow from a received packet. More generally, one could characterize a worm by the topology of the causal graph resulting from the received packet. In fact, one can view the topology or membership of a packet's forward causality graph as a signature of a worm and raise an IDS alert whenever one sees this signature. Similarly, one can view the topology or membership of a network service's information-flow graph as a profile of that network service and raise an alarm whenever one sees an anomalous causality graph.

Finally, in some cases one can take advantage of application-specific knowledge to identify more precisely which incoming packet causes a given action. For example, an email client may be able to inform the information-flow tracking system of which email message contains the attachment that is being viewed. If the attachment compromises the viewer, the information-flow tracking system can focus on the appropriate message.

5.2 Tracking multi-hop attacks

Many organizations place most of their computers and services on an intranet behind a firewall, with only a few computers and services exposed to the public Internet. Relying too heavily on this type of perimeter defense has well-known flaws: if an attacker breaches the firewall (such as through an infected laptop, email virus, or bug in one of the public network services), he or she gains access to the computers on the local network, which are often less secured. To clean up after such a breach, the administrator must first find all the computers that have been compromised. It can be quite difficult to find which computers have been compromised during an attack, because the attacker may break into a system, steal or corrupt data, then clean up by removing telltale signs of their attack.

In this section, we show how BDB tracks attacks that traverse multiple hosts, even when the intrusion is detected initially on only a single host. After an intrusion is detected on a single host, BDB tracks backward to determine what led to the intrusion on that host. Tracking back continues across nodes until it reaches the point at which the attack entered the intranet. Finding the point the attack entered the intranet is useful in securing the network against future attacks. If the attack entered via an infected laptop, the administrator can chastise and educate the user; if the attack entered via a buggy network service, the administrator can disable that service or apply a patch [70]. After backward tracking is complete, BDB performs forward tracking to find other compromised hosts. As BDB finds compromised hosts during backward and forward tracking, it provides the opportunity to learn about the attack. The information it learns can be used to develop network and host signatures of the attack that can be used to avoid future compromises or scan other networks for compromised hosts.

As discussed in Section 5.1, prioritizing which packets to follow can be done in a variety of ways. In Section 5.2.1, we track the Slapper worm backward using the *highest process, most recent packet* heuristic. During the forward tracking phase, we prioritize which packets to follow by leveraging the signatures learned about the worm while tracking it backward. In Section 5.2.2, we prioritize the packets by following those flagged as suspicious by the Snort network IDS. In Section 5.2.3, we use application-specific instrumentation to identify which e-mail message led to later events on the host.

5.2.1 Slapper worm

We first demonstrate the ability of BDB to track attacks across multiple hosts by releasing a modified version of the Slapper worm on a local worm testbed (Figure 5.1). We modified the spreading pattern of Slapper to look for new victims on the local network before using its default method of searching for vulnerable machines among randomly generated IP addresses.

Our testbed contains four vulnerable web servers: one accessible from both the public

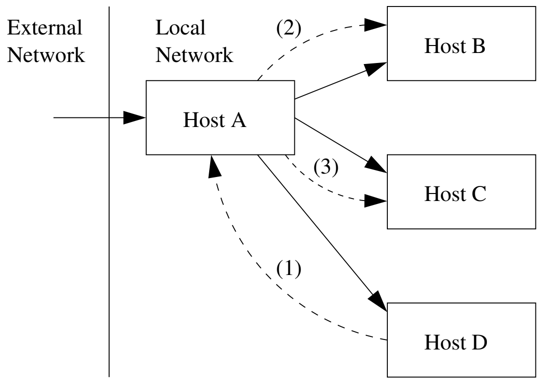


Figure 5.1: Inter-host propagation of the Slapper worm

The worm first infects Host A, then launches attacks against Hosts B, C, and D. After an IDS detects the attack on Host D, BDB tracks the attack backward to Host A, then tracks it forward to Hosts B and C. The solid lines depict the attacks, and the dotted lines depict the order of BDB's analysis.

Internet and the local network, and three accessible only from the local network. To make it harder to track the worm, we add background noise by running the SPECweb99 benchmark between the web servers. Two machines acted as SPECweb99 servers, and the other two machines acted as SPECweb99 clients. This workload generated 95,943 operating system objects and 1,628,937 events spread out over the four machines. The test was run for 20 minutes, with the worm being released 10 minutes after the test began.

The initial detection point occurred on Host D when an IDS detected a suspicious process named *update*; Figure 5.2 shows the information-flow graph on Host D that led to this process. The *update* process is used by Slapper to spread the worm to other machines. This process on Host D is the starting point of our backward traversal. The information-flow graph shows that the worm compiled *update* from source code, which it downloaded by a shell over a socket. The worm spawned the shell from a compromised web server.

Using the *highest process, most recent packet* heuristic, BDB identifies a packet received by the web server process as likely to be part of the attack. That packet originated from Host A and is the starting point for backward tracking on that host. Tracking the packet on Host A results in sequence of events similar to those found on Host D. However, the *highest process, most recent packet* heuristic identifies a packet that originated from an external source, so BDB's traversal backward stops.

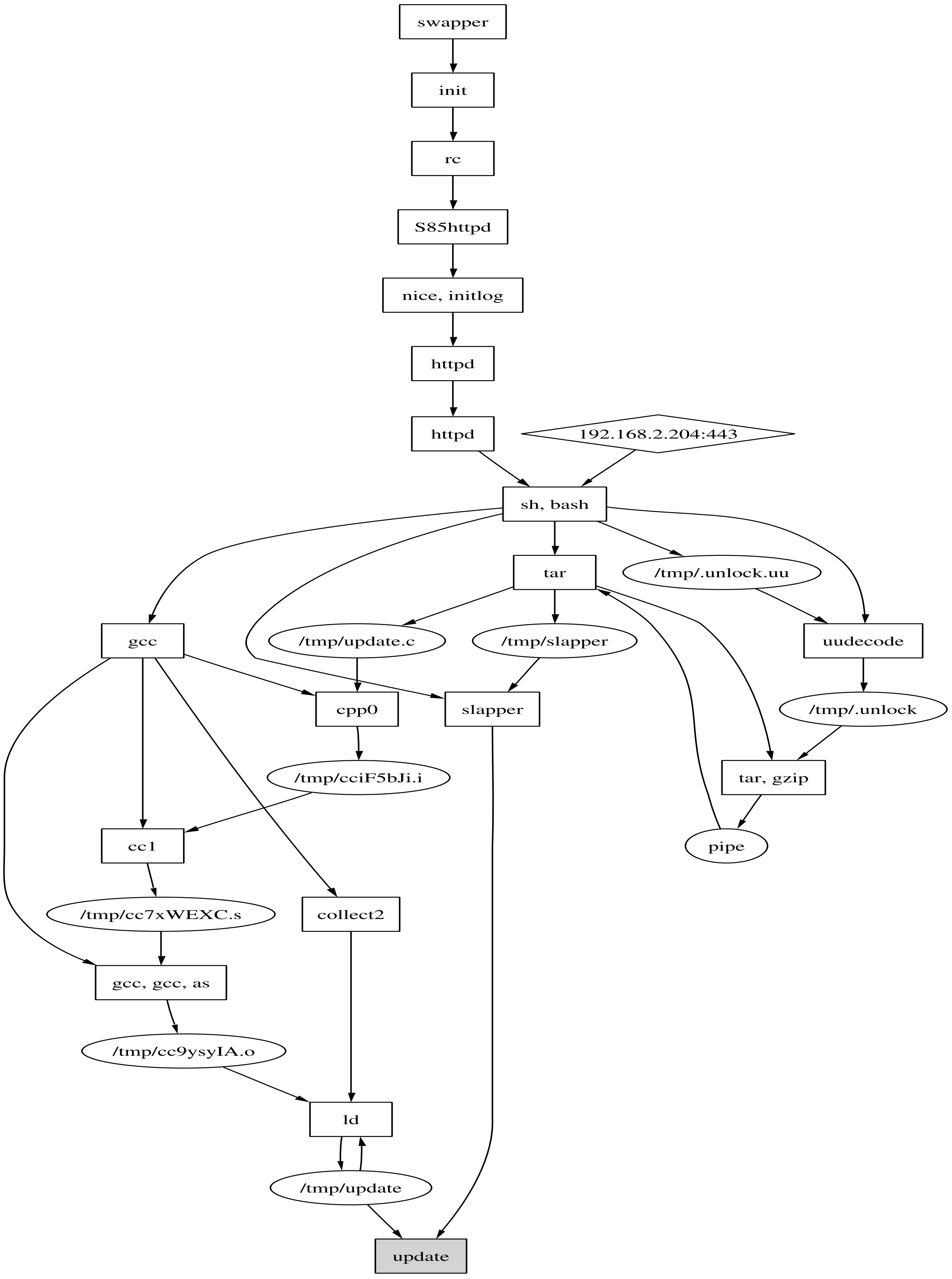


Figure 5.2: Backward information-flow graph for Slapper worm for Host D.

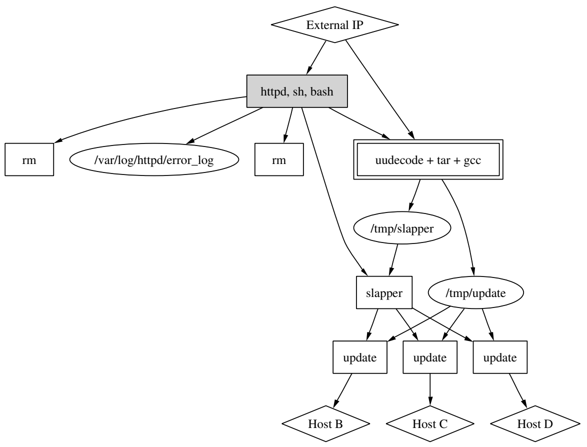


Figure 5.3: Forward causality graph for the Slapper attack

Forward causality graph of Host A for the Slapper attack. The subgraph formed by executing uudecode, tar, and gcc is represented by the box with double lines.

While tracking the worm backward from Host D to Host A, BDB collects information about the worm in the form of a backward information-flow graph. These graphs are signatures that describe how the worm behaves on a computer. BDB uses this information to go forward and detect other instances of the worm.

The forward traversal begins with the httpd process on Host A since that is the starting point of the worm (Figure 5.3). BDB initially considers all activity resulting from this point. In addition to communicating with Host D, the worm also contacts Hosts B and C. BDB uses forward tracking to examine the effects of each outgoing packet on the receiving hosts. To determine if the worm had infected these machines, BDB searches the forward graphs for the information-flow signatures found while traversing backward. In both cases, BDB finds the signature, and the hosts are deemed exploited.

BDB tracks the Slapper worm effectively using the *highest process, most recent packet* heuristic and matching host worm signatures found during the backward traversal phase. Despite the heavy load on the network service that was exploited, BDB filtered out irrelevant events and highlighted only the actions of the worm. The backward and forward graphs of all of the infected nodes contain a total of 171 objects and 251 events, which is 2-3 orders of magnitude less than the total activity on the system.

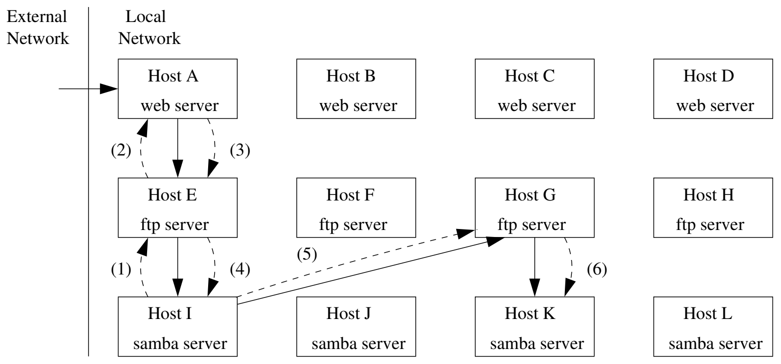


Figure 5.4: Inter-host propagation of the multi-vector attack

We carried out an attack on five of the 16 testbed hosts (A, E, I, G, K) using multiple attack vectors (solid lines). After an IDS detects the attack on Host I, BDB tracks the attack backward to Host E and A, then tracks it forward to Hosts E, I, G, and K.

5.2.2 Multi-vector manual attack

Although the *highest process, most recent packet* heuristic is effective for the Slapper worm and many other existing exploits we examined, an attacker that knows about this heuristic could evade detection. For example, an attacker could implicate an innocent request by receiving it between the initial exploit and the next phase of the attack. Another difficult scenario occurs when there are multiple network services at different roots of the backward graph; in this case, there is no unique highest process.

One way to address these shortcomings is to use a network IDS to prioritize which packets to follow. While a network IDS may not be accurate enough to track attacks by itself, it may be accurate enough to prioritize which packets BDB should follow. In this paper, we use the network IDS Snort [1] to prioritize packets for BDB. Snort matches messages against a number of known signatures; for this test, we use rules that find commonly used shellcode instructions. For example, Snort detects consecutive no-op instructions, which are used often in buffer overflow attacks to compensate for variations in process address spaces. Although this type of network anomaly detector is prone to false positives, it is effective enough to prioritize which causally-related network packets to examine first.

To evaluate whether a network IDS can prioritize packets effectively, we carried out an attack on our local network that exploits several network services on various nodes within our system (Figure 5.4). We simulate a stealthy attacker by compromising one host at a time, rather than scanning the network and attacking all vulnerable hosts. We start by breaking into a publicly accessible web server. From there, we compromise succeeding hosts through vulnerabilities in the ftp and samba servers. Each attack results in an interactive shell, and we use that shell to download tools and break into the next host.

Our testbed includes 12 hosts: four web servers, four ftp servers, and four samba servers. Each of the nodes also doubles as a client. To make it harder to track the attack, we add background noise through artificial users who are logged into each computer. Each user mounts all four samba servers in their file space. The users download source code from a randomly selected web or ftp server, then unpack this source code to a randomly selected samba mounted directory and compile it. The entire process is repeated for the duration of the test. These activities result in large amounts of noise; across all twelve nodes there are over 2 GB of network traffic, 6,589,526 operating system events and 814,262 operating system objects. We ran the test for 20 minutes, and the intrusions occurred 10 minutes after the test started.

The attack is detected initially on Host I, when the attacker launches a backdoor process that opens a raw socket. BDB generates a backward graph using the *backdoor* process as the detection point, then sees which of the causally related incoming packets were flagged as suspicious by Snort. In this case, one of the incoming packets in the backward graph had been flagged as suspicious by Snort. The suspicious packet came from the ftp server on Host E. Using the suspicious packet as the starting point for another BackTracker iteration, BDB again found that one of the causally related packets on Host E's backward graph had been flagged as suspicious by Snort. This packet led us back to the public web server on Host A. All causally related incoming packets on Host A are from external connections, so the backward traversal ends here.

Starting from the external web server, BDB uses Snort to prioritize among the causally related outgoing packets, with the goal of finding other compromised hosts. The forward analysis led us back to Host E and then again to Host I. From Host I, BDB found that the attacker broke into Host G and then Host K.

In the end, BDB found all of the infected hosts and highlighted 420 operating system objects and 19 network packets of the 814,262 objects and 2 GB of network data on the entire system. BDB's resulting information-flow graph is small enough that an administrator who wants to understand the attack in more detail can examine each object and packet by hand.

Although BDB found all compromised hosts, there were some false positives in our analysis. In particular, Snort flagged as high priority seven packets that unsuccessfully attempted to use the samba exploit. These did not affect our analysis because none of the unsuccessful break-in attempts generated any extraneous network activity, and all of the attacked hosts were eventually broken into. To reduce these false positives, BDB could prioritize network packets further by using additional IDSs; for instance, it could see which suspicious network packets led to other host IDS alerts on the receiving host.

5.2.3 E-mail virus

Attacks often propagate through a network via e-mail viruses. E-mail viruses typically spread by fooling a user into running a suspicious application or by exploiting a helper process used to handle an attachment. Unfortunately, the structure of e-mail handling programs makes it difficult for BDB to track the causal relationship between incoming messages and subsequent actions. E-mail servers that receive messages and e-mail clients that display them for users tend to be long-lived and to read multiple messages at a time. Because BDB tracks information flow at the granularity of a process, it assumes conservatively that all messages that have been read affect all subsequent actions. Tracking information flow at a more precise granularity requires program-specific instrumentation, and this section demonstrates how one can enhance BDB in this way to track e-mail viruses.

We made two application-specific changes to support e-mail tracking across multiple hosts. First, we modified the exim e-mail server to link the network messages it receives with e-mail messages IDs. Second, we modified the pine e-mail client to inform BDB of the "current" e-mail message ID being read at a given time. We assume that the saving of an attachment or launching of a helper process is caused by the current e-mail message being read by the user. These changes allow us to track (backward or forward) the effects of an e-mail message as it is received by the server, transferred to the client, and viewed

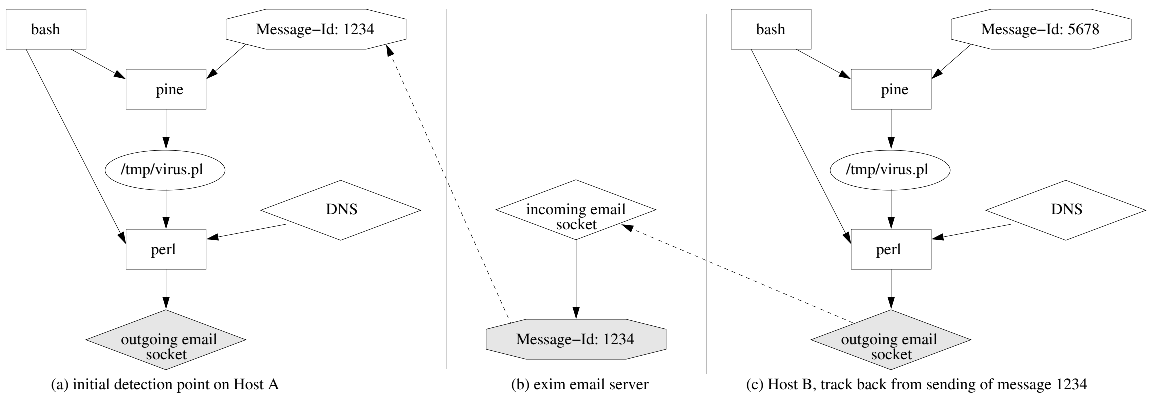


Figure 5.5: Tracking an e-mail virus

In (a), BDB uses its normal tracking plus an instrumented e-mail client to track back from an outgoing network connection to the causally related incoming e-mail message. In (b), BDB uses the instrumented e-mail server to link the suspicious e-mail message with the incoming network message that delivered that e-mail message. This procedure identifies the host that sent the e-mail virus, where the backward tracking can be repeated.

by a user. Resulting actions on a host, such as writing files, starting new processes, and sending messages, are tracked via BDB's normal mechanisms.

We tested the ability of BDB to track an e-mail virus by installing our instrumented e-mail server and client on our testbed, then releasing a virus. Figure 5.5 shows how BDB's resulting information-flow graph tracks the virus backward from the initial detection on Host A (Figure 5.5a), back to when the virus was received by the exim server (Figure 5.5b), back to the upstream Host B (Figure 5.5c).

5.3 Correlating IDS alerts

In this section we discuss the benefits of correlating multiple IDS alerts using information-flow graphs, and we show how correlating alerts with causality can reduce false positives on a testbed system. In this section when we discuss false positives, we are referring to the more traditional use of the term that describes IDS alerts that trigger when no attack is present.

A major problem with many IDSs is false positives. One approach to reduce these false positives is to combine multiple host or networks IDS alerts into a single, higher-confidence alert. Prior approaches connect distinct alerts through statistical correlation on various features of the alert, such as the destination IP address or time of the alert. BDB makes it possible to correlate alerts in a new way by revealing which alerts are related causally.

The main benefit of relating alerts causally is its potential for increased accuracy—statistical correlation may suffer from coincidental events, whereas causal relationships are determined by chains of specific OS and network events.

In addition, using information flow to correlate alerts can reduce dramatically the amount of data each IDS needs to process. For example, in Section 5.2.2, we showed how BDB revealed which outgoing network packets were causally related to a host IDS alert. BDB can thus narrow the search for suspicious network activity to a handful of packets, even in the midst of a busy network. In the same way, BDB can reduce the amount of data that a host IDS must examine. For example, a host IDS need examine only those processes and files that are related causally to packets that are flagged as suspicious by a network IDS. This allows one to use intensive host IDSs that would otherwise be too slow [68].

Two IDS alerts may be related causally in a variety of ways. One alert may causally precede or follow the other. Or, two alerts may share a common ancestor, such as when a single shell process executes two child processes that each perform a suspicious action. It is up to higher-level policies to determine how to score these different causal relationships, and we leave this to future work. We speculate that alerts are more likely to be correlated if one is the direct descendant of the other than if they simply share a common ancestor.

To test how effectively causality can be used to correlate alerts, we ran BDB on two test systems that were exposed to the Internet. One system was running the default RedHat 6.2 installation; the other was running the default RedHat 7.0 installation. Both systems had several vulnerable servers that were accessible from the Internet, including the bind, ftp, and web servers. We configured the system to use a network IDS and a host IDS. We used Snort with its default rules as a network IDS, and we used a host IDS that flagged as suspicious any process that runs as root. While both IDSs are expected to generate many false positives, correlating alerts from these two IDSs using causality can increase our confidence that alerts result from actual compromises. To correlate alerts, we started with

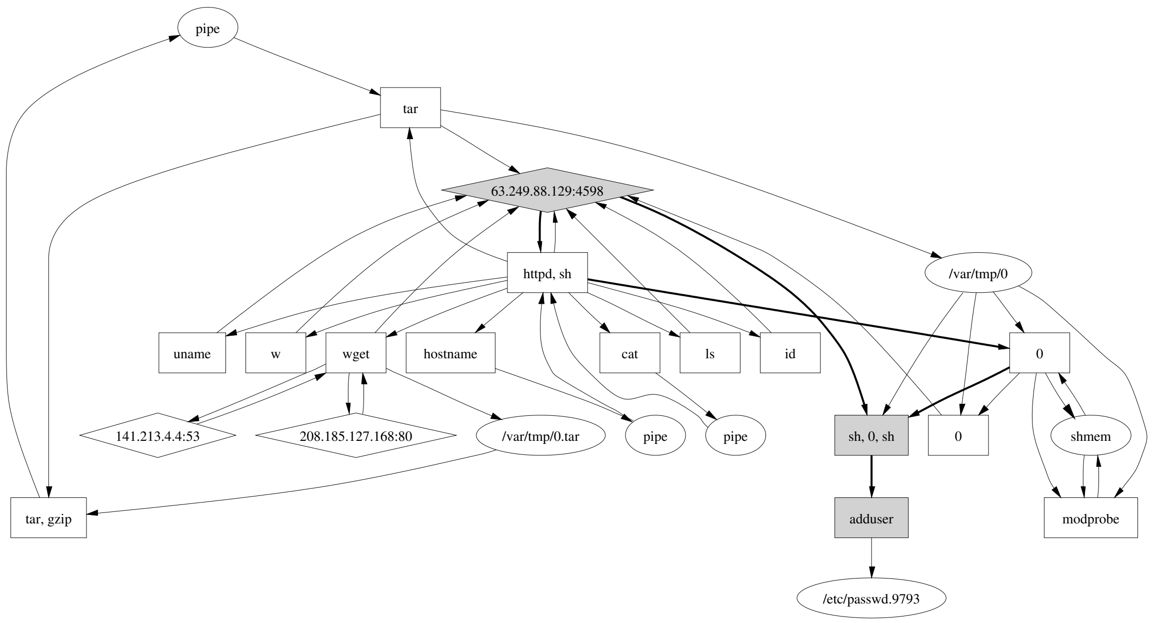


Figure 5.6: Correlating alerts

This figure shows how an information-flow graph can connect two distinct IDS alerts. The two alerts are shown as shaded boxes. One alert is generated by Snort, and the other is generated by a host IDS that detects root processes. The bold lines highlight the events that most directly relate the flagged IDS alerts.

Snort alerts and generated causal graphs on the host for each suspicious network message. We then reported any processes in the resulting information-flow graphs that ran as root.

We ran the system for two days. [1] During this period, Snort generated 39 alerts, and the host IDS detected numerous root processes. Of these alerts, BDB detected two that were connected causally: an outgoing message flagged by Snort and two root processes. Figure 5.6 shows that the alerts were connected causally through a common ancestor (the httpd process), we confirmed by hand that these two alerts were indeed the result of a successful attack. Snort's alert resulted from a suspicious outgoing packet (triggering the "uid=" rule that indicates the output of the id command) that was sent by a shell process executed by a compromised web server. The shell process also downloaded various tools and gained root access through a local *modutils* exploit, which triggered the host IDS alert.

We examined the other Snort alerts by hand to see if there were any other true positives.

[1]We also used BDB on one of our desktop computers to correlate Snort and host IDS alerts. Although we were not attacked successfully over this period, BDB was effective at filtering out the numerous false positives generated by Snort.

We found one other true positive, in which the attacker compromised the web server but did not gain root privilege. This illustrates the tradeoffs involved in correlating IDS alerts. Reporting only correlating alerts reduces the number of false positives, but it may mask some true positives as well. Correlating Snort alerts with a more sophisticated host IDS may have reported both true positives.

Overall, BDB effectively reduced the number of false positives on our testbed. It also reduced administrative overhead by allowing us to use Snort's default rules, rather than customizing the set of rules by hand to reduce false positives.

CHAPTER 6

Limitations

In the prior section, we showed that BackTracker, ForwardTracker, and BDB helped analyze several real attacks. In this section, we consider what an intruder can do to hide their actions from our system. An intruder may attack the layers upon which our Event-Logger module is built, use events that EventLogger does not monitor, or hide his actions within large dependency graphs.

An intruder can try to foil GraphGen or EventLogger by attacking the layers upon which analysis or logging depend. One such layer is the operating system. GraphGen's analysis is accurate only if the events and data it sees have their conventional meaning. If an intruder can change the kernel (e.g., to cause a random system call to create processes or change files), then he can accomplish arbitrary tasks inside the machine without being tracked by EventLogger. Many operating systems provide interfaces that make it easy to compromise the kernel or to work around its abstractions. Loadable kernel modules and direct access to kernel memory (/dev/kmem) make it trivial to change the kernel. Direct access to physical memory (/dev/mem) and I/O devices make it easy to control applications and files without using the higher-level abstractions that EventLogger tracks. Operating systems can disables these interfaces [42]. Operating systems may also contain bugs that allow an intruder to compromise it without using standard interfaces [13]. Researchers are investigating ways to use virtual machines to make it more difficult for intruders to compromise the guest operating system [38].

Another layer upon which one current implementation of EventLogger depends is the virtual-machine monitor and host operating system. Attacking these layers is considerably more difficult than attacking the guest kernel, since the virtual-machine monitor makes the trusted computing base for the host operating system much smaller than the guest kernel.

If an intruder cannot compromise a layer below EventLogger, he can still seek to stop GraphGen from analyzing the complete chain of events from the detection point to the source of the attack. The intruder can break the chain of events tracked if he can carry out one step in his sequence using *only* low-control events that EventLogger does not track. Section 3.4 explains why this is relatively difficult.

An intruder can also use a hidden channel [52] to break the chain of events that EventLogger tracks. For example, an intruder can use the initial part of his attack to steal a password, send it to himself over the network, then log in later via that password. GraphGen can track from a detection point during the second login session up to the point where the intruder logged in, but it cannot link the use of the password automatically to the initial theft of the password. GraphGen depends on knowing and tracking the sequence of state changes on the system, and the intruder's memory of the stolen password is not subject to this tracking. However, GraphGen will track the attack back to the beginning of the second login session, and this will alert the administrator to a stolen password. If the administrator can identify a detection point in the first part of the attack, he can track from there to the source of the intrusion.

An intruder can also try to hide his actions by hiding them in a huge dependency graph. This is futile if the events in the dependency graph are the intruder's actions because the initial break-in phase of the attack is not obfuscated by a huge graph after the initial phase. In addition, an intruder who executes a large number of events is more likely to be caught.

An intruder can also hide his actions by intermingling them with innocent events. GraphGen includes only those events that potentially affect the detection point, so an intruder would have to make it look as though innocent events have affected the detection point. For example, an intruder can implicate an innocent process by reading a file the innocent process has written. In the worst case, the attacker would read all recently written files before changing the detection point and thereby implicate all processes that wrote those files. As usual, security is a race between attackers and defenders. GraphGen could address this attack by filtering out file reads if they are too numerous and following the chain of events up from the process that read the files. The attacker could then implicate innocent processes in more subtle ways, etc.

Finally, an attacker can make the analysis of an intrusion more difficult by carrying out the desired sequence of steps over a long period of time. The longer the period of attack, the more log records that EventLogger and GraphGen have to store and analyze. In conclusion, there are several ways that an intruder can seek to hide his actions from BackTracker. Our goal is to analyze a substantial fraction of current attacks and to make it more difficult to launch attacks that cannot be tracked.

CHAPTER 7

Enhancing Operating-System-Level Information Flow

It may be difficult for system administrators to analyze intrusions using large information-flow graphs. Information-flow graphs can become large by including objects that interact with many distinct OS-level objects, or by including OS-level objects that accumulate taint conservatively, causing false positives. In this chapter we discuss techniques that can be used to reduce the size of large OS-level information-flow graphs. Two key principles guide our discussion: (1) separate the level of granularity used to display an information-flow graph from the level of granularity used to determine causally connected events and objects, and (2) identify stateless objects and enforce the stateless property to reduce false positives.

Separating the level of granularity used to display an information-flow graph from the level of granularity used to determine casually connected events and objects might help reduce the size of large information-flow graphs. The key to this principle is the observation that both course-grained information flow and fine-grained information flow help reduce the size of information-flow graphs, but in different ways. Course-grained information flow consolidates objects, which helps with the visualization aspect of an information-flow graph. But, course-grained information flow can cause false positives in an information-flow graph, which adds more objects and contradicts our goal of making the graph smaller. Fine-grained information flow reduces false positives by tracking information flow through subcomponents of course-grained objects. But, fine-grained information flow adds more objects and events to the system, which the information-flow graph displays. To resolve these conflicting attributes, we recommend using more course-grained information flow for displaying information-flow graphs, but more fine-grained information flow to track dependencies.

Identifying stateless objects and enforcing the stateless property can reduce the size of information-flow graphs by eliminating false positives. Stateless objects are objects that service incoming events deterministically, independent of any previous events. Our current system taints objects for the entire duration of the object. We do not have a mechanism for determining when an event no longer affects an object, so long-lived objects accumulate taint without the ability to clear dependencies, even for stateless objects. To eliminate these false dependencies, we can identify stateless objects and prune unrelated events.

In this chapter, with these two principles in mind, we discuss how one can reduce the size of OS-level information-flow graphs using course-grained information flow, using advanced OS-level techniques, and using fine-grained information flow. This chapter only discusses techniques for reducing the size of information-flow graphs, we did not implement any of these techniques.

7.1 Course-grained information flow

Course-grained information flow may help reduce the size of OS-level information-flow graphs by grouping similar OS-level objects into a single *composite object.* Grouping objects into a composite object hides any events between OS-level objects within the composite object, giving the system administrator less data to analyze and reducing the size of the information-flow graph. Reducing the size of the information-flow graph eliminates some information that is normally available to the system administrator; composite objects must have semantic meaning so the system administrator still has enough information to analyze the intrusion. For example, a system administrator can consolidate all web server objects into a single composite object that encapsulates web server processes, cgi scripts, content files, and log files. This consolidation hides the internal operations of the web server, but the administrator can still attribute actions on the system to the web server. Composite objects should be used for visualization only, system administrators should still track OS-level events and objects within the composite object itself. By tracking OS-level objects within composite objects, administrators can reduce many of the false positives that arise by tracking at the more coarse level of granularity.

To reduce the size of an OS-level information-flow graph further, a system administrator can identify stateless composite objects to reduce false positives. Stateless composite objects are groups of processes and files that deterministically process events independent of any previous events. Enforcing the stateless property for composite objects by simply disallowing internal state changes is impractical since processes naturally evolve as they run, and temporary files are used often. Fortunately, system administrators can use checkpoint and rollback techniques [31] to enforce the stateless property for composite objects. Rolling back composite objects removes all taint accumulated over the checkpoint interval. Removing taint disconnects distinct events associated with a composite object, unless the events occur over the same checkpoint interval. If events occur over the same checkpoint interval, we correlate them only when they connect via OS-level events and objects within the composite object.

When using checkpoint and rollback to enforce the stateless property for composite objects, the main difficulty is finding the appropriate time to take the checkpoint without affecting computations. Rolling back a composite object restores the internal state of the object, which clears the taint. However, if an ongoing computation uses that internal state, rolling back the state might corrupt the computation. For example, a web server can be divided into a front-end composite object and a file server composite object. The front end is stateless and handles user requests, and the file server component maintains state and stores persistent web-server objects. The system administrator can rollback the front-end composite object periodically to disconnect unrelated requests, but if they perform the rollback in the middle of handling a request, the request might become corrupted or get dropped because it relies on state held within the front-end object. Addressing this difficulty requires domain specific techniques that take into account the semantics of the computations carried out by the composite object.

7.2 Advanced OS-level techniques

Programmers can use fork system calls to handle concurrent requests and to reduce false positives. Servers typically handle concurrent requests using multiple entities that can be scheduled, the two most common used in today's operating system are processes (created using fork) and kernel threads. Using fork, servers are essentially stateless with a single parent process that waits for new connections and creates child processes to handle the new connections. The forked child begins by copying the parent's address space, so the parent's state is effectively the checkpoint and forking a new process rolls back to that checkpoint. A forked child depends on the parent, but the parent does not depend on the child because the child has a private copy of the parent's address space. As a result, requests are causally disconnected, except through the single parent process. Using threads, the flow of the application is the same, except new requests are handled by a child thread. The child thread shares the parent's address space for increased performance, but this sharing causes all address-space changes made by the child to be visible to the parent and all other children. As a result, the parent and all children are causally connected through the shared address space, which causally links all requests, including requests without true dependencies. Fortunately, we configured all servers used in our experiments (i.e., http, ftp, smb) to use fork.

System administrators may be able to configure threaded servers to be stateless using checkpoint and rollback techniques. However, they encounter the same problems as they do with composite objects where finding a place to take a checkpoint is challenging and all requests handled within a checkpoint interval are connected. Furthermore, checkpoint and rollback implementations use the same mechanisms as fork (e.g., page protections) typically, so administrators lose any performance gains from using threads when enforcing the stateless property on a threaded server.

7.3 Fine-grained information flow

A system administrator can supplement OS-level information flow with fine-grained information flow to reduce excessive tainting. For performance reasons, it is infeasible to use fine-grained information flow for the entire system. However, an administrator can identify key OS-level objects that are stateless and tend to accumulate excessive taint, and use fine-grained techniques to eliminate false dependencies for these objects. Fine-grained information flow techniques (i.e., PL-level or instruction level) divide OS-level objects into subcomponents. By tracking subcomponents individually we hope to eliminate false dependencies that arise from tainting entire OS-level objects. Furthermore, we can track the effects of these fine-grained objects and try to determine when they no longer affect tainted objects.

A programmer can add annotations to a program to certify certain data as low-control. This technique of certifying data is similar to classifying OS-level events as high-control and low-control, but it is done by the programmer using program specific constructs instead of being done by the administrator using OS-level constructs. Certifying data works by specifying invariants within a program and registering this specification with the OS. The OS can verify the invariants to classify the data as low-control. This technique may reduce the size of information flow graphs, and it leverages knowledge of the programmer, who has intimate domain specific knowledge of the program and how it handles data. However, these annotations introduce a new place for programmers to make mistakes, and a mistake certifying data may remove key information from an information-flow graph, making it difficult to analyze the intrusion.

CHAPTER 8

Future Work

This dissertation focuses on using information-flow graphs to analyze intrusions, but it might also be possible to use information-flow graphs to detect or prevent intrusions. Information-flow graphs can be used to describe behavior on a computer system. A graph of *known-good* behavior defines how a service interacts with the system and how information flows between components, any deviations from this graph of known-good behavior may indicate potentially suspicious activity. In addition, a graph of *known-bad* behavior defines known malicious activities, acting as a signature for an intrusion detection system. This chapter discusses these additional uses for information-flow graphs.

Host-based intrusion detection systems (IDS) view events and objects in isolation. A localized view of events and objects can be problematic because it does not take into account how one object affects another. As a result, IDSs may be too imprecise to detect intrusions. For example, consider the Windows local security authentication server (lsass.exe) and the Windows services daemon (services.exe). These two processes communicate using a named pipe. Services.exe can install new system daemons, but when services.exe installs a new daemon in response to a command sent by lsass.exe, it is probably the sign of an intrusion. The problem is not that services.exe installs a new daemon or that lsass.exe sends commands to services.exe through a named pipe; those are both typical actions performed independently by both processes. The problem is that lsass.exe initiates the action when functionally, it has no reason to install a new system service. In general, many IDSs [41, 59, 50, 19, 7] take into account only objects directly associated with specified processes or files without considering the global flow of information. OS-level information flow graphs show the system-wide effects of causal events and objects, and can be used to specify known-good behavior on a host. By defining known-good behavior, it may be possible to detect and prevent intrusions by stopping anomalous deviations from this known-good behavior. We anticipate that this approach can be effective for a wide range of Windows and Linux services and the implementation will be suitable for real-time detection and prevention.

It might be possible to use information-flow graphs as a way of defining known-bad behavior and using them as a signature for malicious activity. One difficulty in using information-flow graphs as signatures is that most of the actions captured in a malicious information-flow graph result from an attack. Thus, attackers can evade this type of signature easily by changing their actions on a compromised host. However, under certain assumptions, information-flow graphs can be useful for creating signatures of known-bad behavior. Specifically, if you assume that the attacker breaks into a network service, then uses the compromised host to propagate the attack to an additional host, information flow graphs can capture this behavior. Semantically, the top of an information flow graph represents system processes, like network services. The bottom of an information flow graph shows the actions of an attacker, including attack propagation. Since there are a limited number of network services that can be broken into on a host, and a limited number of network services that can be used to propagate the attack, the top and the bottom of an information flow graph can act as a signature for malicious behavior. This relationship associates various distinct activities on a host. For example, information-flow graphs can show that a web server process causes another process to start making web requests (i.e., propagating the attack), which is probably indicative of malicious activities.

CHAPTER 9

Related Work

9.1 Intrusion analysis

BackTracker tracks the flow of information [26] across operating system objects and events. The most closely related work is the *Repairable File Service* [79], which also tracks the flow of information through processes and files by logging similar events. The Repairable File Service assumes an administrator has already identified the process that *started* the intrusion; it then uses the log to identify files that potentially have been contaminated by that process. In contrast, BackTracker begins with a process, file, or filename that has been affected by the intrusion, then uses the log to track back to the source of the intrusion. The two techniques are complementary: one could use backtracking to identify the source of the intrusion, then use the Repairable File Service's forward tracking to identify the files that potentially have been contaminated by the intrusion. However, we believe that the forward tracking technique has fundamental limitations, as we discussed in Section 4.3.

In addition to the direction of tracking, EventLogger and GraphGen differ from the Repairable File Service in the following ways: (1) EventLogger tracks additional dependency-causing events (e.g., shared memory, mmap'ed files, pipes and named pipes; (2) GraphGen labels and analyzes time intervals for events, which are needed to handle aggregated events such as loads/store to mmap'ed files; and (3) GraphGen uses filtering rules to highlight the most important dependencies. Perhaps most importantly, we use EventLogger and GraphGen to analyze real intrusions and evaluate the quality of the dependency graphs it produces for those attacks. The evaluation for the Repairable File Service has so far focused on time and space overhead—to our knowledge, the spread of contamination has been evaluated only in terms of number of processes, files, and blocks contaminated and has been performed only on a single benchmark (SPEC SDET) with a randomly chosen initial process.

Work by Goel *et al.* uses techniques outlined in this dissertation to automate intrusion recovery with a system called *Taser* [40]. Taser tracks OS-level information flow to identify objects that have been affected by an attack, and automatically removes tainted data while leaving untainted data in place. In our work, to recover from intrusions, we provide information-flow graphs to administrators and rely on the administrators to manually recover from the intrusion based on the data within the information-flow graph. Another difference between this work and Taser is that Taser recovers from intrusions only on single hosts, whereas BDB tracks attacks across multiple hosts.

Work by Ammann, Jajodia, and Liu tracks the flow of contaminated transactions through a database and rolls back data if it has been affected directly or indirectly by contaminated transactions [10].

Several other projects assist administrators in understanding intrusions. CERT's Incident Detection, Analysis, and Response Project (IDAR) seeks to develop a structured knowledge base of expert knowledge about attacks and to look through the post-intrusion system for signs that match an entry in the existing knowledge base [23]. Similarly, SRI's DERBI project looks through system logs and file system state after the intrusion for clues about the intrusion [65]. These tools automate common investigations after an attack, such as looking for suspicious filenames, comparing file access times with login session times, and looking for suspicious entries in the password files. However, like investigations that are carried out manually, these tools are limited by the information logged by current systems. Without detailed event logs, they are unable to describe the sequence of an attack from the initial compromise to the detection point.

Security audit logs [11] have been studied since the 1980's, but recent projects improved the state-of-the-art in security audit logs. The "perfect" security audit log is complete (i.e., capture *all* information), can guarantee integrity, and add little overhead. ReVirt

[29] and Flight Data Recorder [67] implements come closer to meeting these goals. ReVirt creates security audit logs for entire machines by logging non-deterministic events beneath the operating system, at the virtual-machine-monitor level. ReVirt uses these logs to deterministically replay the execution of the entire virtual machine, including the operating system and all applications. During replay, system administrators can monitor the state of the system as Joshi *et al.* [43] do with the IntroVirt project. Flight Data Recorder implements OS-level logging in Windows Vista systems, and does so efficiently enough to handle server production servers at Microsoft's MSN division. Both ReVirt and Flight Data Recorder are complete and provide strong integrity (assuming the virtual-machine monitor is trusted for ReVirt and OS is trusted for Flight Data Recorder), and add little overhead to the system. Early prototypes of EventLogger were tested using ReVirt, and Flight Data Recorder logs may be suitable replacements for EventLogger logs.

9.2 Information flow

We use information flow to help system administrators analyze intrusions, but information flow has been used since the 1970's with many different applications.

Traditional information-flow systems aim to prevent the flow of sensitive information to untrusted sources [52]. The Bell LaPadula model [14] implements multilevel security [2] using mandatory access controls to prevent secrets from being leaked in a military setting. They introduce two key axioms that prevent principles from "reading up" from higher security classes, or from "writing down" to lower security classes, and thus preventing the leakage of sensitive information from higher security classes to lower security classes. Denning extends this work by providing a lattice model of information flow [26] to prevent information leakage. Denning's model focuses on the flow of information rather than access control decisions made at individual objects, and her model provides an elegant theoretical framework that many subsequent projects use. More recently, Myers and Liskov [55], and Zdancewic *et al.* [77] track information flow at the programming language level to prevent leakage of sensitive information. They allow users to declassify their own data, thus providing a decentralized model for information flow control. Efstathopoulos *et al.* [30] extend this idea of users declassifying data and apply it to OS-level objects in the

Asbestos operating system. Yumerefendi, Mickle, and Cox [76] use a form of process-level replay to automatically detect when sensitive information leaks across the network. They detect leaked information by modifying sensitive data, sending the modified data to a cloned process, then they replay the clone to determine if the replaying clone's network output changes. If the output changes, then sensitive information is leaked.

Recent projects use fine-grained information flow through a program to determine if data is being used improperly. Newsome and Song [57], and Kiriansky *et al.* [48] use fine-grained information flow techniques on assembly instructions to determine if user-supplied data flows to sensitive areas, like function pointers or return addresses located on stacks. Suh *et al.* [62] also use similar techniques, but introduce hardware modifications needed to make this type of fine-grained information flow tracking efficient. Xu *et al.* [75] track fine-grained information flow by transforming C source code, rather than working at the assembly level. Chow *et al.* track instruction-level information flow, but do so to determine where sensitive data flows throughout a system [21] and propose solutions to preventing this type of flow [22].

The Perl programming language tracks the flow of tainted information across perl program statements [69]. Like the Repairable File Service, Perl tracks the forward flow of contaminated information rather than backtracking from a detection point to the source of the intrusion.

Causality can help debug programs and distributed applications. One technique is program slicing [64] which determines which execution paths and variables of a program affect a certain portion of an application, presumably where the effects of a bug were detected. Program slicing can be performed statically [74] to determine all variables and paths that *could* have affected the current state, or dynamically [8] to determine which variables and paths affected the current state for the most recent execution. Our techniques could be viewed as OS-level dynamic program slicing where OS-level objects are our variables and the entire machine is our program. A second technique also used for debugging is proposed by Aguilera, *et al.* [9]. In this work they infer causal relationships based on timing of messages passed between distributed nodes, and use these causal relationships to debug performance issues.

Lamport uses causal messages to synchronize local clocks on distributed nodes [51]. These messages establish a “happens before” relationship between the nodes that sets a partial ordering of events on the nodes.

9.3 Multi-hop attacks and correlating IDS alerts

In addition to our approach of using information flow, other techniques have been used to track multi-hop attacks. Work by Zhang and Paxson detects “stepping stones” used to carry interactive communication across multiple hops by seeing which packets have correlated size and timing characteristics [78]. Wang and Reeves extend this approach by manipulating the timing of packets to more easily correlated packets across multiple hops [71]. Our approach has advantages and disadvantages compared to prior approaches. The main disadvantage is that our approach requires one to monitor each host in the chain. Hence our approach is suitable mainly for communication within a single administrative domain. The main advantage of our approach is that it is independent of timing because it can track actual cause-and-effect chains, rather than relying on less robust characteristics such as timing. Hence our approach can track non-interactive multi-hop attacks, such as worms (even stealthy ones).

Many prior researchers have sought to correlate IDS alerts to reduce false positives [66] [25] [58]. Most prior projects correlate IDS alerts based on shared features, such as source or destination IP addresses of packets or the time at which the IDS alert occurred. Other projects use prior knowledge about sequences of actions to connect IDS alerts together into an attack scenario. Our work adds a new way to connect IDS alerts, by tracking actual cause-and-effect chains to connect prior alerts with later ones.

9.4 Sandboxing IDSs

Sandboxing IDSs are closely related to the idea of using information-flow graphs for intrusion detection because they both monitor similar events (i.e., system calls). Janus [41], to our knowledge, was the first to propose using these techniques. Systrace [59] expands on Janus by proposing novel techniques for profiling services and handling false positives. Recursive Systrace [50], BlueBoX [19], and MAPbox [7] extend sandboxing rules to child and helper processes. Using information-flow graphs for intrusion detection improves upon these projects by taking into account the global flow of information, and utilizing additional information not monitored by Sandboxing IDSs. These improvements may result in more precise specifications of services.

CHAPTER 10

Conclusions

In this dissertation we showed that operating-system-level information-flow graphs were effective at helping to analyze intrusions. We developed techniques for tracking OS-level information flow on local hosts and tracking OS-level information flow across multiple hosts. EventLogger used this information-flow tracking to generate logs that were used by GraphGen to reconstruct past states and events, and to help analyze intrusions. Starting from a detection point, such as a suspicious file or process, BackTracker identified the events and objects that could have affected that detection point. The dependency graphs generated by BackTracker helped administrators find and focus on a few important objects and events to understand the intrusion. Using these dependency graphs, the administrator can identify the process that was most likely to be the source of the intrusion; this process became the starting point for the forward analysis. ForwardTracker started from the original source of the intrusion and followed the attack forward identifying a set of objects and events that were likely to have resulted from the attack. BDB continued the analysis across the network by following attacks across multiple hosts.

We used our techniques to analyze several real attacks against computers we set up as honeypots, and also to analyze several artificial workloads designed to test the limits of our techniques. In each case, BackTracker was able to highlight effectively the entry point used to gain access to the system and the sequence of steps from the entry point to the point at which we noticed the intrusion. ForwardTracker was able to show the attacker's activities after the break in, and BDB highlighted the set of computers that were also part of the attack.

Our techniques were effective at highlighting the activities of an attacker because we started the analysis with single object that we believed, with a high level of confidence, was part of an attack. Because we began with a malicious object, it was likely that the objects that affected this malicious object, and the objects that were affected by this malicious object, were also malicious. As a result, we were able to process audit logs that contained mostly benign events and objects, and we highlighted a subset of the events and objects that were likely to be part of the attack.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Snort: The Open Source Network Intrusion Detection System. http://www.snort.org/.

[2] Department of Defense Trusted Computer System Evaluation Criteria (orange book). Technical report, Department of Defense, December 1985. DoD 5200.28-STD.

[3] Microsoft security bulletin ms03-026, 2003. http://www.microsoft.com/technet/security/bulletin/MS03-026.mspx.

[4] Symantec security response – w32.blaster.e.worm, 2003. http://www.symantec.com/avcenter/venc/data/w32.blaster.e.worm.html.

[5] Microsoft security bulletin ms04-011, 2004. http://www.microsoft.com/technet/security/Bulletin/MS04-011.mspx.

[6] Trend micro virus encyclopedia – worm_agobot.gy, 2004. http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=WORM_-AGOBOT.GY.

[7] Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[8] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990. ACM Press.

[9] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[10] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from Malicious Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, September 2002.

[11] James P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co., April 1980. Contract 79F296400.

[12] Charles J. Antonelli, Matthew Undy, and Peter Honeyman. The Packet Vault: Secure Storage of Network Data. *Proc. USENIX Workshop on Intrusion Detection and Network Monitoring*, April 1999.

[13] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 131–147, May 2002.

[14] David E. Bell and Leonard LaPadula. Secure computer system:unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., 1976.

[15] Kerstin Buchacker and Volkmar Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the 2001 IEEE Symposium on High Assurance System Engineering (HASE)*, pages 95–105, October 2001.

[16] CERT. Steps for Recovering from a UNIX or NT System Compromise. Technical report, CERT Coordination Center, April 2000. http://www.cert.org/tech_tips/win-UNIX-system_compromise.html.

[17] CERT. Detecting Signs of Intrusion. Technical Report CMU/SEI-SIM-009, CERT Coordination Center, April 2001. http://www.cert.org/security-improvement/modules/m09.html.

[18] CERT. Multiple Vulnerabilities In OpenSSL. Technical Report CERT Advisory CA-2002-23, CERT Coordination Center, July 2002. http://www.cert.org/advisories/CA-2002-23.html.

[19] Suresh N. Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003.

[20] Bill Cheswick. An Evening with Berferd in Which a Cracker is Lured, Endured, and Studied. In *Proceedings of the Winter 1992 USENIX Technical Conference*, pages 163–174, January 1992.

[21] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 2004 USENIX Security Symposium*, August 2004.

[22] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[23] Alan M. Christie. The Incident Detection, Analysis, and Response (IDAR) Project. Technical report, CERT Coordination Center, July 2002. http://www.cert.org/idar.

[24] CIAC. L-133: Sendmail Debugger Arbitrary Code Execution Vulnerability. Technical report, Computer Incident Advisory Capability, August 2001. http://www.ciac.org/ciac/bulletins/l-133.shtml.

[25] Frederic Cuppens and Alexandre Miege. Alert Correlation in a Cooperative Intrusion Detection Framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[26] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[27] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[28] Lynn Doan. College door ajar for online criminals. *Los Angeles Times*, May 2006. http://www.latimes.com/news/printedition/la-me-hacks30may30,1,6143368.story.

[29] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.

[30] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazi&#232;res, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM Press.

[31] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3), September 2002.

[32] Dan Farmer. What are MACtimes? *Dr. Dobb's Journal*, (10), October 2000.

[33] Dan Farmer. Bring out your dead. *Dr. Dobb's Journal*, (1), January 2001.

[34] Dan Farmer and Wietse Venema. Forensic computer analysis: an introduction. *Dr. Dobb's Journal*, (9), September 2000.

[35] Fedral Bureau of Investigation. 2005 fbi computer crime survey, January 2006. http://www.fbi.gov/publications/ccs2005.pdf.

[36] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing, January 1998. RFC 2267.

[37] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, pages 120–128, 1996.

[38] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)*, February 2003.

[39] Sharon Gaudin. Sobig 'carpet bombs' the internet. *ClickZNews*, August 2003. http://www.clickz.com/news/article.php/3066881.

[40] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal del Lara. The taser intrusion recovery system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[41] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Security Symposium*, pages 1–13, July 1996.

[42] Xie Huagang. Build a secure system with LIDS, 2000. http://www.lids.org/document/build_lids-0.2.html.

[43] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP)*, pages 91–104, October 2005.

[44] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. In *Proceedings of 1994 ACM Conference on Computer and Communications Security (CCS)*, pages 18–29, November 1994.

[45] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, pages 223–236, October 2003.

[46] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Technical Conference*, pages 71–84, June 2003.

[47] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, February 2005.

[48] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 2002 USENIX Security Symposium*, August 2002.

[49] Stefan Kornexl, Vern Paxson, Holger Dreger, Anja Feldmann, and Robin Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *Proceedings of the 2005 Internet Measurement Conference (IMC)*, October 2005.

[50] Aleksey Kurchuk and Angelos D. Keromytis. Recursive sandboxes: Extending systrace to empower applications. In *Proceedings of the 19th IFIP International Information Security Conference (SEC)*, August 2004.

[51] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[52] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[53] John Leyden. Virus floors russian stock exchange. *The Register*, February 2006. http://www.theregister.co.uk/2006/02/03/virus_hits_stock_exchange.

[54] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI)*, June 2005.

[55] Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the 1997 Symposium on Operating Systems Principles*, October 1997.

[56] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. In *Proceedings of the 2003 Workshop on Runtime Verification (RV'03)*, July 2003.

[57] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, February 2005.

[58] Peng Ning, Yun Cui, and Douglas S. Reeves. Constructing Attack Scenarios through Correlation of Intrusion Alerts. In *Proceedings of the 2002 ACM Conference on Computer and Communications Security (CCS)*, pages 245–254, November 2002.

[59] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 2003 USENIX Security Symposium*, pages 257–272, August 2003.

[60] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, pages 295–306, August 2000.

[61] Staff Writer. Computer worm grounds flights, blocks atms. CNN.com, Jaunary 2003. http://www.cnn.com/2003/TECH/internet/01/25/internet.attack/.

[62] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 2004 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[63] The Honeynet Project, editor. *Know your enemy: revealing the security tools, tactics, and motives of the blackhat community.* Addison Wesley, August 2001.

[64] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[65] W. M. Tyson. DERBI: Diagnosis, Explanation and Recovery from Computer Break-ins. Technical Report DARPA Project F30602-96-C-0295 Final Report, SRI International, Artificial Intelligence Center, January 2001. http://www.dougmoran.com/dmoran/publications.html.

[66] Alfonso Valdes and Keith Skinner. Probabilistic Alert Correlation. In *Proceedings of the 2001 International Workshop on the Recent Advances in Intrusion Detection (RAID)*, October 2001.

[67] Chad Verbowski, Emre Kiciman, Arunvijay Kumar, Brad Daniels, Shan Lu, Juhan Lee, Yi-Min Wang, and Roussi Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *Proceedings of the 2006 Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[68] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of 2001 IEEE Symposium on Computer Security and Privacy*, 2001.

[69] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, 3rd edition*. O'Reilly & Associates, July 2000.

[70] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, August 2004.

[71] X. Wang and D. S. Reeves. Robust Correlation of Encrypted Attack Traffic Through Stepping Stones by Manipulation of Interpacket Delays. In *Proceedings of the 2003 ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[72] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeyper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *Proceedings of the 18th Large Installation System Administration Conference (LISA)*, November 2004.

[73] Bernhard Warner. Home pcs rented out in sabotage-for-hire racket. *USA Today*, July 2004. http://www.usatoday.com/tech/news/computersecurity/2004-07-07-zombie-pimps_x.htm.

[74] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[75] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, August 2006.

[76] Aydan Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Duke University Technical Report CS-2006-07*, June 2006.

[77] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: secure program partitioning. In *Proceedings of the 2001 Symposium on Operating Systems Principles*, October 2001.

[78] Yin Zhang and Vern Paxson. Detecting Stepping Stones. In *Proceedings of the 2000 USENIX Security Symposium*, August 2000.

[79] Ningning Zhu and Tzi-cker Chiueh. Design, Implementation, and Evaluation of Repairable File Service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN)*, pages 217–226, June 2003.

ABSTRACT

Analyzing Intrusions Using Operating System Level Information Flow

by

Samuel T. King

Chair: Peter M. Chen

Computers continue to get broken into, so intrusion analysis is a part of most system administrators' job description. System administrators must answer two main questions when analyzing intrusions: "how did the attacker gain access to my system?", and "what did the attacker do after they broke in?". Current tools for analyzing intrusions fall short because they have insufficient information to fully track the intrusion and because they cannot separate the actions of attackers from the actions of legitimate users.

We designed and implemented a system for analyzing intrusions by using OS-level information flow to highlight the activities of an attacker. OS-level information flow is a collection of causal events which connect operating system objects. These causal events can be linked to form an information-flow graph which highlights the events and objects that are part of an attack.

Information flow graphs can be used to help system administrators determine how an intruder broke into a system and what they did after the compromise. We developed

BackTracker to determine how an intruder broke into a system. BackTracker starts with a suspicious object (e.g., malicious process, trojaned executable file) and follows the attack back in time, using causal events, to highlight the sequence of events and objects that lead to the suspicious state. Showing a graph of these causally-connected events and objects provides a system-wide view of the attack and significantly reduces the amount of data an administrator must examine in order to determine which application was originally exploited. We also developed ForwardTracker to determine the attacker's actions after the compromise. ForwardTracker starts from the application which was exploited and tracks causal events forward in time to display the information flow graph of events and objects that result from the intrusion. Furthermore, we designed and implemented Bi-directional Distributed BackTracker (BDB) which continues the backward and forward information flow graphs across the network to highlight the set of computers on a local network which are likely to have been compromised by the attacker.