UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Model of Forensic Analysis Using Goal-Oriented Logging

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Sean Philip Peisert

2007

The dissertation of Sean Philip Peisert is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2007

For Kathryn — my wife, my inspiration, my everything.

And for my parents, who gave me what I needed to get here.

SHERLOCK HOLMES: *"It is an old maxim of mine which states that once you have eliminated the impossible, whatever remains, however improbable, must be the truth."*

—Sir Arthur Conan Doyle, "The Sign of the Four,"
Lippincott's Monthly Magazine (1890)

DEDUCE (verb): *draw as a logical conclusion*

New Oxford American Dictionary, Second Edition (2005)

LOGIC (noun): *The art of thinking and reasoning in strict accordance with the limitations and incapacities of human misunderstanding.*

—Ambrose Bierce, *The Devil's Dictionary* (1911)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

> While Sherlock Holmes was not a doctor, Conan Doyle had based Holmes's method as a detective upon one of his former professors of medicine at Edinburgh University, Dr. Joseph Bell, whose powers of observation and deduction had made him a wizard at diagnosis.
>
> With the exception of Edgar Allan Poe's pioneering stories about Auguste Dupin of Paris, Dr. Conan Doyle recalled many years later, most contemporary fictional detectives produced their results by chance or luck. Dissatisfied with that, he had decided, he said, to create a detective who would treat crime as Dr. Bell had treated disease. This meant, in short, the application of scientific method to crime detection. That was a novel concept in 1887, to be sure, but it worked, first in fiction and then in practice, with life imitating art as it so often does when the art in question is a work of genius.
>
> ...
>
> As a detective in a scientific sense, Holmes always wants to know and looks for the physical evidence; he made himself a master at observing and analyzing physical evidence that the police and other detectives overlook or fail to recognize at all. By his innate but also rigorously trained powers of deduction, he is able to reason backwards from this evidence to reconstruct the crime and delineate the physical attributes of the perpetrator. ... Holmes pays little attention to the psychology of crime ... [1]

This passage describes a 120-year-old methodology for analyzing crime successfully, that started as fiction and became reality. Amazingly, these words still apply today to computer forensic analysis. However, until now, computer forensic analysis has largely been performed in the same way that Holmes's fictional predecessors, and Scotland Yard's real predecessors, performed forensic analysis: by chance or luck. Sometimes, chance and luck are enough. However, it is no coincidence that the most famous pieces of computer detective work have been performed by unusually brilliant computer analysts, such as Bill Cheswick [Che92], Cliff Stoll [Sto88, Sto89], and Tsutomu Shimomura [Shi95, SM96, Shi97]. But there are more attacks and attackers in the world than genius cyberdetectives available to analyze those attacks.

It will always *help* to be a a genius cyberdetective to analyze computer crime, and even for them, luck will never hurt. However, our goal is to change the level to which one must rely on these things by using a rigorous method of analyzing *facts* rather than relying on luck, chance, or what we think we might know about an intruder's abilities, psychology, or motives. In this way, even the non-genius cyberdetective has a little more of a chance of getting things right.

In this dissertation, we describe the year 2007 application of these year 1887 ideas ...

[1] See the afterword ("Dr. Kreizler, Mr. Sherlock Holmes ...") by Jon Lellenberg, in [Car05].

ACKNOWLEDGEMENTS

Many people have given me generous support and encouragement during my life, my time as a Ph.D. student, and during the process of writing this dissertation.

I would like to thank my advisors, teachers, mentors, and coaches — Sid Karin, Matt Bishop, Larry Carter, and Keith Marzullo. They have guided me through a series of steps in my life and my work that I could not have made it through without them — and have become friends in the process. In the best way that I can, I hope to live up to the gifts that they have given me. The advice from my entire dissertation committee has been interesting and valuable. I appreciate their interest, support, and guidance, and I hope to have the opportunity to continue to interact with all of them in the future.

Special thanks to Becky Bace, Martha Dennis, Drew Gross, Tsutomu Shimomura, Abe Singer, and Kevin Walsh, who all gave me support at important times and in important ways, and who also taught me new ways to think about academia, careers, and computer security.

I wish to thank Robert S. Cohn and Steven Wallace at Intel for their enhancements to the FreeBSD version of the dynamic, binary instrumentation tool, *Pin*, which greatly helped my research on forensic analysis using sequences of function calls.

Finally, I would like to thank my patient and wonderful wife, Kathryn (who also copy-edited this entire dissertation); my closest friends, Aaron, Greg, Kent, Laura, Noah, PJ, and Stephen; and all of my family, who have all given me support throughout my life, have helped to make this possible, and have ultimately made the end result mean much more than just obtaining a degree.

This material is based on work sponsored in part by: the Air Force Research Laboratory under Contract F30602-03-C-0075, a Lockheed-Martin Information Assurance Technology Focus Group 2005 University Grant, and award ANI-0330634, "Integrative Testing of Grid Software and Grid Environments," from the National Science Foundation.

The following papers, which have been previously published or are currently in submission, are reprinted in this dissertation with the full permission of all co-authors of the papers:

- "Principles-Driven Forensic Analysis," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, in *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, pp. 85–93, Lake Arrowhead, CA, October 2005.

- "Your Security Policy is *What??*" Matt Bishop and Sean Peisert, *University of Calfornia at Davis Technical Report*, CSE-2006-20, October 2006.

- "Toward Models for Forensic Analysis," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, to appear in *Proceedings of the 2nd International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*, Seattle, WA, April 2007.

- "Analysis of Computer Intrusions Using Sequences of Function Calls," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, conditionally accepted with minor revisions by *IEEE Transactions on Dependable and Secure Computing (TDSC)*, January 2007.

VITA

1999	Bachelor of Arts in Computer Science Minor in Organic Chemistry University of California, San Diego
2000	Master of Science in Computer Science University of California, San Diego
2000–2001	Co-Founder and Chief Operating Officer Dyna-Q Corporation
2000–2001	Visiting Scholar San Diego Supercomputer Center
2002	Lecturer Department of Computer Science and Engineering University of California, San Diego
2002–2003	Assistant Director California Next-Generation Internet Applications Center
2004–2005	Project Technical Lead California Internet2 Technology Evaluation Center
2002–2005	Computer Security Researcher San Diego Supercomputer Center
2005	Candidate in Philosophy in Computer Science University of California, San Diego
2005–2007	Graduate Student Researcher Department of Computer Science and Engineering University of California, San Diego
2001–Present	Fellow San Diego Supercomputer Center
2007	Doctor of Philosophy in Computer Science University of California, San Diego

PUBLICATIONS

"A Programming Model for Automated Decomposition on Heterogeneous Clusters of Multiprocessors," Sean Philip Peisert, M.S. Thesis, March, 2000.

"Forensics for System Administrators," Sean Peisert, *;login: The Magazine of USENIX*, vol. 30, no. 4, August, 2005, pp. 34–42. (Reprinted in *Cyber Forensics: Tools and Practices*, ICFAI University Press, 2007.)

"Principles-Driven Forensic Analysis," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, in *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, pp. 85–93, Lake Arrowhead, CA, October 2005.

"Your Security Policy is *What??*" Matt Bishop and Sean Peisert, *University of Calfornia at Davis Technical Report*, CSE-2006-20, October 2006.

"Analysis of Computer Intrusions Using Sequences of Function Calls," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, conditionally accepted with minor revisions by *IEEE Transactions on Dependable and Secure Computing (TDSC)*, January 2007.

"Toward Models for Forensic Analysis," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, to appear in *Proceedings of the 2nd International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*, Seattle, WA, April 2007.

PAPERS IN SUBMISSION

"How to Design Computer Security Experiments," Sean Peisert and Matt Bishop, submitted to the *Fifth World Conference on Information Security Education (WISE)*, West Point, NY, February 2007.

FIELDS OF STUDY

Major Field: Computer Science
Studies in Computer Security.
Professors Sid Karin, Matt Bishop, and Keith Marzullo

Studies in Performance Programming and Complexity Theory.
Professor Larry Carter

Major Field: Music
Studies in Conducting.
Professor Thomas Nee,
Living Room Conservatory, Leucadia, California

ABSTRACT OF THE DISSERTATION

A Model of Forensic Analysis Using Goal-Oriented Logging

by

Sean Philip Peisert
Doctor of Philosophy in Computer Science
University of California, San Diego, 2007
Professor Sidney Karin, Chair

Forensic analysis is the process of understanding, re-creating, and analyzing arbitrary events that have previously occurred. It seeks to answer such questions as how an intrusion occurred, what an attacker did during an intrusion, and what the effects of an attack were.

Currently the field of computer forensics is largely *ad hoc*. Data is generally collected because applications log it for debugging purposes or because someone thought it to be important. Practical forensic analysis has traditionally traded off analyzability against the amount of data recorded. Recording less data puts a smaller burden both on computer systems and on the humans that analyze them. Not recording enough data leaves analysts drawing their conclusions based on inference, rather than deduction.

This dissertation presents a model of forensic analysis, called *Laocoön*, designed to determine what data is necessary to understand past events. The model builds upon an earlier model used for intrusion detection, called the *requires/provides model*. The model is based on a set of qualities we believe a good forensic model should possess. Those qualities are in turn influenced by a set of five principles of computer forensic analysis. We apply Laocoön to examples, and present the results for a UNIX system. The results demonstrate how the model can be used to record smaller amounts of highly useful data, rather than forcing a choice between overwhelming amounts of data or such a small amount of data to be effectively useless.

1

Introduction

Sherlock Holmes: *"It is of the highest importance in the art of detection to be able to recognize out of a number of facts which are incidental and which vital. Otherwise your energy and attention must be dissipated instead of being concentrated."*

—Sir Arthur Conan Doyle, "The Adventure of the Reigate Squire," *The Strand Magazine* (1893)

1.1 Background

Forensic analysis is the process of understanding, re-creating, and analyzing arbitrary events that have previously occurred. It seeks to answer questions of *how* an intrusion occurred, *what* the attacker did during the intrusion, and *what effects* the attacker's actions had. *Logging* is the recording of data that will be useful in the future for understanding past events. *Auditing* involves gathering, examining, and analyzing the logged data to understand the events that occurred during the incident in question.

Forensic analysis is a cornerstone of security and accountability. Analysts use it to recover lost, damaged, or deleted data, and to determine what actions created the problems that led to the loss. Analysts also attempt to associate these actions with the users who executed them, for the purposes of accountability. Forensics has two goals. The more common one is to derive a past state (or sequence of states) or events from the current state, log data, and system context. This goal includes associating users with events that cause state transitions, specifically actions that damage system or data integrity. The less common goal is to take a sequence of state transitions and determine events that are likely to follow those events. This is useful when the traces of an attack are found and the analysts attempt to determine what other actions the attacker(s) may have taken.

1.2 Problem Statement

Currently the field of computer forensics is largely *ad hoc*. Data is generally collected because applications log it for debugging purposes (UNIX `syslog` [All05]) or because someone thought it to be important (BSM [ON01]). Further, the two elements of forensic analysis, logging and auditing, are completely divorced from each other. That is, the system administrators selecting the data to be logged, work independently from — and have very different goals than — the forensic analysts who ultimately analyze the data. The result is that too much of the wrong data is currently being recorded by most forensic systems, rendering analysis difficult or impossible. There is typically an enormous amount of information to analyze; determining which entries in the log correspond with the intrusion can be hard, and trying to correlate log entries can be very difficult to do. In the end, the analyst may never be certain about how the intrusion took place, and have to be satisfied with guessing about vulnerabilities to patch.

Practical forensic analysis has traditionally traded off accuracy against the amount of data recorded. One can hope to help the analyst by recording information on more events. This can indeed help. But this is only a partial solution, and at some point adding more logged information can be counterproductive. There are several reasons for this. For example, sometimes, logged data can be redundant. At other times, logged data could be superfluous and unrelated to the incident in question, therefore reducing accuracy of the eventual analysis by a human by generating too much information that must later be identified and removed. Finally, sometimes logged data can be misleading, such as in the cases where logged data is intentionally generated by malicious programs and is intentionally designed to divert an analyst's attention.

A forensic solution at one extreme [PBKM05] is to record everything. This would include all memory accesses explicitly made in an intruder's program, rather than those added as intermediate storage by the compiler. Another answer [Gro97] along the same lines is recording everything that causes a system to transition from one state to another. The other end of the spectrum is to record very high-level (and unstructured [Bis95]) data such as syslog messages or data that is focused in one particular area. Examples of this include filesystem data from *Tripwire* [KS94], other file auditing systems [Bis88a], and *The Coroner's Toolkit* [FV]; or connection data from *TCP Wrappers* [Ven92]. Finally, there have been attempts [PBKM07a] to find a middle ground, but those attempts still try to solve forensic problems by recording and analyzing *available* data. A rigorous solution for finding and recording the *necessary* data to perform forensic analysis is needed.

1.3 Thesis Statement

In this dissertation, we present a model and a methodology designed to determine what data is necessary to log for the purpose of forensic analysis in a way that preserves the ability to analyze how an attack occurred, what an attacker did during the intrusion, and what the effects of an attack were. By recording only the relevant data, the approach can be more efficient in resource usage, and more accurate in an eventual analysis, than existing approaches.

1.4 Approach

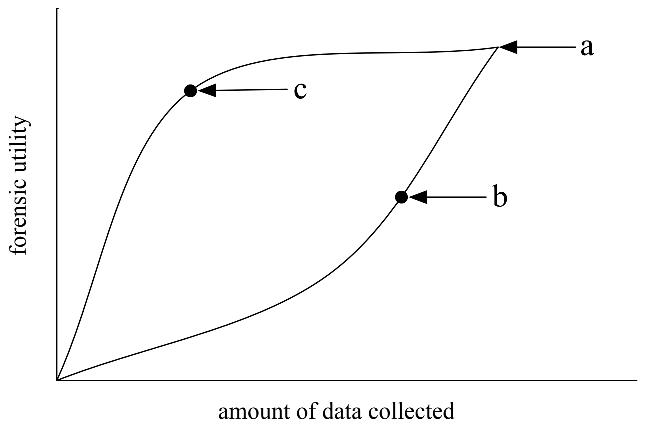


Figure 1.1: Diagram of possible measures of utility based on different data collected. (a) represents logging everything and analyzing everything [PBKM05]. (b) represents less logging than (a) but with comparable utility [PBKM07a]. (c) represents our goal, by using a model of forensics, in this dissertation.

Our goal is to take steps toward formalizing forensic analysis. To do so, in this dissertation, we seek to determine what data is necessary to understand past events, and therefore carefully select smaller amounts of highly useful data to record, rather than forcing system administrators to make a choice between recording overwhelming amounts of data or such a small amount of data to effectively be useless. Whereas recording everything, as shown in Figure 1.1a, *might* give maximum *utility*, it is impractical, if not infeasible. Similarly, our earlier approach of recording all function calls [PBKM07a] also gives high utility, but also a substantial cost, as shown in Figure 1.1b. Our goal is to attempt to develop a methodology to optimize the data necessary to record events, perhaps also providing a tuning parameter that allows system administrators and forensic analysts to collect more or less data with correspondingly more or less utility based on individual scenarios. This is shown in Figure 1.1c. The overall utility may be less than shown in Figure 1.1b, but may come close, with vastly less data necessary. But the data actually collected would be only the *relevant* data, rather than the extra data that may be misleading, redundant, or superfluous.

Therefore, it is important to make it clear, in this dissertation, that we focus on the *logging* part of forensics to ultimately make the second part, *auditing*, more tractable. Techniques for good auditing are a distinct line of research, however, and therefore, we leave it for future research.

The absence of a rigorous approach to forensics indicates the need for a *model* from which to extract the exact logging requirements. We are not aware of the existence of such a model. We previously identified five, high-level principles of forensic analysis [PBKM05]. We use these principles, as well as several qualities [PBKM07b] that we believe a good forensic model should possess, to guide the construction of a model.

In this dissertation, we describe a formal model of critical elements of forensic analysis, and use that model to derive what needs to be recorded to perform effective forensic analysis with consideration of the cost in performance and other measures. We discuss a method of capturing more *effective* data rather than simply *more* data, and as a result, reduce the burden on both the computer system and the human analyst. Finally, we demonstrate the effectiveness of the model through empirical data from experiments. This methodology can be used both to harden existing systems and to develop new systems that include support for forensic analysis in the design.

Our model builds upon recent work in formalizing multi-stage attacks for the purposes of intrusion detection [TL00, ZHR$^+$07], to develop formalisms that will allow us to derive rigorously what is needed to log for forensic analysis. This approach should not only provide insight into the current practices of forensic analysis, but should also provide guidance for designers of new systems that will make forensic analysis simpler and more effective.

To model attacks to determine the information necessary to record in order to perform forensic analysis, we need to know the *goal-oriented actions* taken by an intruder and record enough information to be able to understand the result. The result might be success or failure, or might be more subtle or less "binary" in nature. We do this by modeling the end result from one or more starting points, as well as possible intermediate states that are common to many or all paths near the start of the attack and near the end. The reason that we do this is due to convergence of methods, at the start (Figure 1.2a) and end of an attack (Figure 1.2c/d), as opposed to the explosion of possible methods that could be used in the middle of an attack (Figure 1.2b).

The key assumption that we make in this dissertation is that our forensic software

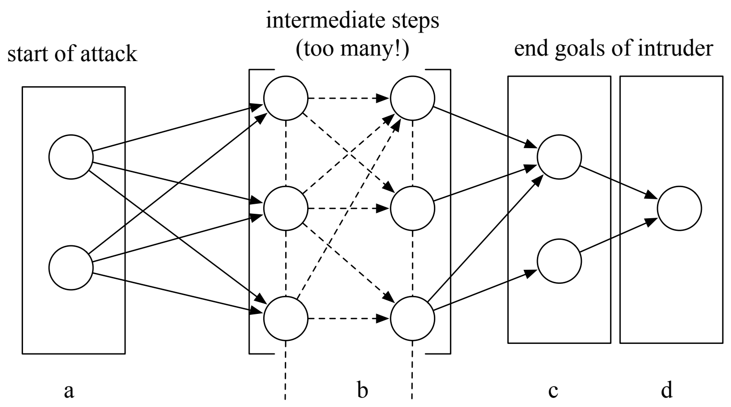


Figure 1.2: Diagram of a generic attack where circles represent actions. An attack model almost always consists of at least the endpoint (d), but may also include the beginnings (a) and possibly other states near the end (c).

obtains accurate information from the system, and is able to report that information correctly [Tho84]. Also note that this dissertation contains a number of references to UNIX-like systems and security tools on UNIX-like systems. However, the use of UNIX-like examples in this document is done for simplicity and consistency of the explanation, and the model can be applied to machines running other operating systems, as well.

The model is one part of our forensic process. The entire forensic process involves the following steps:

1. Determine what data to use as inputs to our model. In this dissertation, we do this manually. However, in Section 8.2, we describe our ideas on automating the process based on security policies.

2. Use the model, described in Chapter 5, to determine what information from the computer system is necessary to log. We show examples of this process in Chapter 6.

3. Instrument the system to collect the relevant data, as discussed in Chapter 7.

4. Analyze the intrusion using the logged data, as we also discuss in Chapter 7.

1.5 Definitions

We define an *event* that we wish to reconstruct as some action that users (legitimate or illegitimate) can take themselves, or can automate a computer to take by programming and compiling code. In our case, in the examples that we use on a FreeBSD system, this typically means code written in C/C++. This does not include hardware manipulation or events that occur *within* a virtual machine (VM), such as the Java VM. The reason we do not address events within VMs is because we make the simplifying assumption that a program within a VM is dangerous only to other programs in the VM, and therefore it is up to the VM to maintain to safe interactions with other programs running in the same VM instance. However, this does not mean that a VM itself is perfectly safe, and thus, we would need to monitor interactions between a VM and its external environment, such as the operating system, and other programs in the same security domain or higher.

An *attack* is a sequence of events that violates a *security policy* of the site. It may occur externally, as when an attacker breaks in, or internally, as when an authorized user uses resources in an unauthorized way (the *insider problem* [Bis05a, Bis05b]).

The *goal* of the attack is to achieve a particular violation: for example, to read a confidential file, or to obtain unauthorized rights. To achieve a goal, the attacker must initiate a series of events (possibly a unique event, possibly one of a set of events). In some sense, the goal is the output of a black box. An *exploit* is the mechanism that an attacker uses to take advantage of a flaw present in the system. Most attack graphs that involve outsiders becoming insiders, or insiders elevating their privileges consist of at least one exploit. However, unlike goals, the actual exploit is not something that we can always predict. Therefore, our process is designed to use information about the goals that we *can* predict to help us record information about the attack and analyze information about the goals that we *cannot* predict.

An *attack graph*, in this dissertation, is a series of multiple goals linked together. Events towards the end of the graph depend on the results or effects of events that occur earlier in the graph.

We stated earlier that forensic analysis is the process of logging and auditing *arbitrary* events. In this dissertation, we will largely focus specifically on *attacks* (including the insider problem), and not on other elements that are often part of forensic analysis in practice, such as recovering erased data, or legal issues, such as chain of custody of digital evidence for use in court. Those elements are important, and we believe that our work could be applied to them in the future, but again, we leave these topics for future research.

1.6 Organization of the Dissertation

This dissertation is organized as follows: Chapter 2 discusses related work and Chapter 3 presents an approach of addressing forensic analysis using sequences of function calls. Chapter 4 describes the need for forensic models and the qualities that they should possess, and Chapter 5 presents our forensic model that incorporates the qualities previously described. Chapter 6 presents several examples of intruder goals and discusses the resulting information necessary to log, and Chapter 7 discusses the implementation of several of the models and forensic results of doing so. Chapter 8 discusses a set of steps, particularly policy discovery, that we will investigate in the future in order to create a robust, automated implementation of our model. Finally, Chapter 9 presents our conclusions.

2

Related Work

Sherlock Homes: *"It is, of course, a trifle, but there is nothing so important as trifles."*

—Sir Arthur Conan Doyle, "The Man with the Twisted Lip," *The Strand Magazine* (1891)

In practice, forensic analysis generally involves locating suspicious objects or events and then examining them in enough detail to form a hypothesis as to their cause and effect. Data for forensic analysis can be collected by introspection of a virtual machine[1] during deterministic replay [DKC^{+}02], as long as the overhead for the non-deterministic event logging is acceptable, and as long as the target machine is limited to a single-processor.[2] Highly specialized hardware [XBH03, NPC05, CFG^{+}06] might make non-deterministic event logging practical. Most existing tools, however, simply operate on a live, running system.

As mentioned earlier, most of the previous work in forensic analysis, starting with Anderson's first proposed use of audit trails [And80], has been *ad hoc.* For example, Bonyun [Bon80] argued for the use of audit trails, and discussed the merits of certain data and the placement of mechanisms to capture that data, but did not discuss how the process of selecting data could be generalized. Indeed, throughout the early evolution of audit trails, sophisticated logging capabilities were developed for multiple platforms, including a Compartmented Mode Workstation

[1]We said earlier forensic analysis of events *within* a virtual machine is beyond the scope of this dissertation. In this context, we discuss the application of a virtual machine *to* forensic analysis, which is different than analyzing events inside virtual machines.

[2]Virtual machines with deterministic replay capabilities fail on multiprocessor machines, because neither the hypervisors nor the operating systems know the ordering of simultaneous reads and writes, by two or more threads running on different processors, to the same location in shared memory. The order of memory reads and writes is critical for deterministic replay. Also, differences from the original runtime can accumulate upon replay, suggesting that often, the original runtime provides the only authoritative record of what happened.

[CFG$^+$87, Pic87, BPWC90], Sun's MLS [Sib88], and VAX VMM [SM90]. Nonetheless, in each of the latter two cases, the data was chosen in an ad hoc fashion, and the reasons for the choices made in selecting the data was largely unstated [Bis03]; in the former case, the logging mechanism was described, but the data was still left to a system administrator to manually specify in advance.

Today, using syslog entries is one of the most common forensic techniques. However, syslog was designed for debugging purposes for programmers, not security [All05]. Similarly, the popular Sun Basic Security Module (BSM) [ON01] and cross-platform successors are constructed based on high-level assumptions about what "seems" important to security, not what has been rigorously shown to be important to security.

Some of the previous work has still been quite successful, and led to useful tools. The most successful forensic work has involved unifying these tools using a "toolbox" approach [FV04, Pei05] that combines application-level mechanisms with low-level memory inspection and other state-based analysis techniques. Examples of such mechanisms include *Tripwire* [KS94] and *The Coroner's Toolkit* [FV], which record information about files, or *TCPwrappers* [Ven92], which records information about network communications. Ultimately, being able to combine the information from these mechanisms to arrive at an understanding of attacks and events often relies on luck rather than methodical planning. Specifically, an analyst who finds the machine immediately after an intrusion has taken place can gather information from that system's state before the relevant components are overwritten or altered. Without this information, the analyst is forced to reconstruct the state from incomplete logs and the current, different state — and as the information needed to do the reconstruction is rarely available, often the analyst must guess. The quality of reconstruction depends on the quality of evidence present, and the ability of the analyst to deduce changes and events from that information. Often, these tools cannot address a large class of forensic problems because the level of granularity is far too high, the data is difficult to correlate, and, even when used together, they do not look at and record information about enough sources of necessary data to perform a thorough forensic analysis.

The approaches used by these previous tools and research projects show value, but since they were not based on a rigorous model of forensic analysis, frequently miss information or capture and display superfluous information that inflates the amount of storage needed and adds nothing to the forensic analysis. Further, the information that many of these existing tools capture is difficult to correlate between multiple tools.

BackTracker [KC05] uses previously recorded system calls and some assumptions about system call dependencies to generate graphical traces of system events that have affected or have been affected by the file or process given as input. However, an analyst using BackTracker may not know what input to provide, since suspicious files and process IDs are not easy to discover when the analysis takes place long after the intrusion. Unfortunately, BackTracker does not help identify the starting point; it was not a stated goal of BackTracker, or its successors [SV05, KMLC05]. Nor does BackTracker frequently help to identify what happens *within* the process, because BackTracker is primarily aimed at a process-level granularity.

Forensix [GFM$^+$05] also collects system calls, but rather than generating an event graph, it uses a database query system to answer specific questions that an analyst might have, such as, "Show me all processes that have written to this file." Forensix had similar constraints as BackTracker. A forensic analyst, for example, has to independently determine which files might have been written to by an intruder's code.

In addition to ad hoc approaches to forensic analysis, a few approaches have used forensic models. Gross [Gro97] studied usable data and analysis techniques from unaugmented systems. He formalized existing *auditing* techniques already used by forensic analysts. One method that he demonstrated first involved classifying system events into categories representing transitions between system states. Then, he demonstrated methods of analyzing the differences between system states to attempt to reconstruct the actions that occurred between them, using assumptions about the transitions that must have come before. He did not discuss a methodology for separating the relevant information from the rest of the system information. Our goal, in contrast to his, is to address a methodology of understanding the information actually *necessary* to analyze specific, discrete events such as attacks. Gross's research focused on using *available* data that was already being collected on systems to improve analysis, and make it more efficient. Our focus is on augmenting a system to collect *necessary* data that is not yet being collected.

Previous research in modeling systems has also been performed to understand the limits of auditing in general [Bis89] and auditing for policy enforcement [Sch00]. However, neither of these previous research efforts were aimed at presenting *useful* information to a human analyst. They were not specifically aimed at forensic analysis but had different goals, such as process automation. Other modeling work [Kup04] evaluated the effect of using different audit methods for different areas of focus (attacks, intrusions, misuse, and forensics) with different temporal divisions (real-time, near real-time, periodic, or archival), but again, the results focused primarily on performance rather than forensic value to a human.

Data from intrusion detection systems has been proposed for use as legal forensic evidence [SB03], but the papers containing those proposals focus on legal admissibility [Som98] and using intrusion detection systems simply because that data is already collected in real-time [Ste00], and not on the utility of the data collected.

There are two types of auditing: *state-based* and *transition-based* [Bis03]. State-based auditing requires state-based logging. This is performed by periodically sampling the state of the system. This technique is arbitrary, imprecise, and has the risk of slowing a system down too much.[3] Transition-based auditing requires transition-based logging. Transition-based logging involves monitoring events, which are more easily recorded. The reason is is two-fold: first, while both state-based and transition-based logging require deciding in advance on levels of granularity to record, state-based logging also requires making decisions about the frequency in which to save state information, which adds an arbitrary element that we would like to avoid. Second, although state-based logging is used commonly in many areas of computer science, such as debugging (breakpoints) and fault tolerance (checkpoints), both of these tasks can involve an iterative process of logging, analysis, and replay. For example, when a bug is suspected, a series of breakpoints might be inserted to check the values of a number of variables at a point in time. If the variables being checked do not suggest a cause, or if a fault occurs again before the breakpoint is reached, then the breakpoints can be changed and the program re-run to determine whether the new breakpoint might be more helpful in locating the bug. However, in security and forensics, unless deterministic replay is used, there is no possibility to do an exact replay of the series of events that led to a suspected attack. The relevant information is often only seen once. So if an attack occurs in between two breakpoints that capture state information, the attack may or may not be revealed in those snapshots of the system state, because traces of the attack may already have been removed by the time the second snapshot is taken. Therefore, in forensics, one must rely on specific events to trigger the logging mechanism. Thus, transition-based logging is generally the most appropriate.

Our approach models attacks using states and transitions between the states, but merges this technique with a model of attacks. Further, by analyzing the amounts of data collected and system performance, our approach provides information that system administrators and forensic analysts can use to balance performance with both forensic effectiveness and efficiency. We have also previously attempted to find a middle ground, but still approached forensic analysis informally, using available data [PBKM07a]. The approach described in this dissertation builds upon our previous work to formalize computer forensic analysis, to determine methods to limit the amount of data recorded, and to examine the trade-offs of collecting the data with respect to performance, reliability, and the ability to perform the forensic analysis.

In some cases, an implementation of our model could require logging more data than some of the existing approaches. For example, a minimal syslog configuration might record lit-

[3]This has been shown to be effective by using a coprocessor to perform the state-based logging and auditing, and then taking periodic hashes of the kernel's image in memory and comparing them to a known-safe state [PFMA05].

tle to no data at all. In other cases, using our model might record radically less information than other approaches. For example, even a "standard" syslog configuration can generate huge amounts of data. One incident at the San Diego Supercomputer Center [MB05, Sin05] resulted in nearly 3 GB of syslog data for a 1-week period, consisting of 28,634,491 log messages. For certain workloads, ReVirt [DKC$^+$02] has been shown to generate at least 1.4 GB of log data per day, creating up to a 68% processor overhead (including overhead due to virtualization). BackTracker, which captures system calls, has been shown to generate 1.2 GB of log data per day, creating up to a 38% processor overhead (including overhead due to virtualization). Additionally, our examination of function calls has shown that between 0.5% and 5% of function calls recorded are system calls, and thus, the amount of audit data increases from 20 to 200 times as compared to recording only system calls. Finally, we have observed that the number of assembly instructions per system call is generally between 1,000 and 10,000, if one were to capture all assembly instructions.

However, the goal of implementing our model is that only the relevant data is captured, and redundant, superfluous, and misleading data is reduced or avoided. Sometimes assembly instructions will be useful, and at other times, system calls, function calls, or their arguments may be most useful. Therefore, even if the model requires a greater amount of logged data than other approaches (such as a minimal syslog configuration), it will be optimized for the goal of analyzing attacks. For example, 1.2 GB of log data per day may be considered practical, but if only 0.1 GB of that data is actually relevant to analyzing attacks, and further, if for the remaining 1.1 GB of data, a different set of data would have been *more* relevant, then the log data would clearly be considered un-optimized.

Other approaches have had different goals and many were successful in achieving those goals. But those goals were not optimizing the data based on the needs of analysis. As a result, some of the existing approaches have resulted in useful tools that could be employed in our own approach, but since the goals of the existing approaches are different from our own, they are not directly comparable with our own method.

To summarize, some previous work has been successful, but none of it, including the work that has touched on modeling, has focused on making the data as effective for forensic analysis as possible. We can leverage some of the techniques to help understand qualities for what a forensic model should possess, however. We describe some of our own early work in performing forensic analysis using sequences of function calls in the next chapter, as our experience with that approach guided our more recent approach to formalizing forensics.

3

A Method of Forensic Analysis Using Sequences of Function Calls

"Is there any point to which you would wish to draw my attention?"
"To the curious incident of the dog in the night-time."
"The dog did nothing in the night-time."
"That was the curious incident," remarked Sherlock Holmes.

—Sir Arthur Conan Doyle, "Silver Blaze,"
The Strand Magazine (1892)

SHERLOCK HOLMES: *"Circumstantial evidence is a very tricky thing. It may seem to point very straight to one thing, but if you shift your own point of view a little, you may find it pointing in an equally uncompromising manner to something entirely different."*

—Sir Arthur Conan Doyle, "The Boscombe Valley Mystery,"
The Strand Magazine (1888)

The work presented in this chapter was first described in an earlier paper by Peisert, et al. [PBKM07a]. This chapter describes our own early approach to forensics. We developed this approach prior to seeking an approach based on a model. By looking at the results of this approach, we gained valuable insight that helped us develop forensic principles, and ultimately, more formal methods.

3.1 Finding Intrusions

The problem of computer forensics is not simply finding a needle in a haystack: it is finding a needle in a stack of needles. Given a suspicion that a break-in or some other "bad" thing has occurred, a forensic analyst needs to localize the damage and determine how the system was compromised. With a needle in a haystack, the needle is a distinct object. In forensics, the point at which the attacker entered the system can be very hard to ascertain, because in audit logs, “bad” events rarely stand out from “good” ones.

In this chapter, we demonstrate the value of recording function calls to forensic analysis. In particular, we show that function calls are a level of abstraction that can often make sense to an analyst. Through experiments, we show that the technique of analyzing sequences of function calls that deviate from previous behaviors, gives valuable clues about what went wrong.

Forensic data logged during an intrusion should be detailed enough for an automated system to flag potentially anomalous behavior, and descriptive enough for a forensic analyst to understand. While collecting as much data as possible is an important goal [PBKM05], a trace of machine-level instructions, for example, may be detailed enough for automated computer analysis, but is not descriptive enough for a human analyst to interpret easily.

There has been considerable success in capturing system behavior at the system call (sometimes called *kernel call*) level of abstraction. All users, whether authorized or not, must interact with the kernel, and therefore use system calls to perform privileged tasks on the system. In addition, kernel calls are trivial to capture and are low-cost, high-value events to log, as opposed to the extremes of logging everything (such as all machine instructions) or logging too little detail for effective forensic analysis (such as syslog). Capturing behaviors represented at the system call abstraction makes intuitive sense: most malicious things an intruder will do use system calls. Nonetheless, extending the analysis of behaviors to include more data than system calls, by collecting function calls, can produce information useful to a human [Bac00] without generating impractical volumes of data. Though function call tracing is not new,[1] we analyze sequences of function calls in a way that results in improved forensic analysis, which we believe *is* new.

Logging all function calls can generate a huge amount of data. Function calls capture forensically significant events that occur both in user space and kernel space (system calls are, essentially, protected function calls). In our experiments, between 0.5% and 5% of function calls recorded in behaviors are system calls. Thus, the amount of audit data increases from 20 to 200 times as compared to recording only system calls. This additional data makes it much easier to determine when something wrong took place, what exactly it was, and how it happened. Additionally, as we describe later, the increase in the amount of data *recorded* does not necessarily translate into a proportional increase in the amount of data necessary for a human to *audit*.

[1]For example, it was used for profiling as early as 1987 [Bis87], and more recently for intrusion detection [MVVK06].

Our approach comes partially from intrusion detection. The techniques need to be modified to be useful for forensic analysis, but as we show here, they have good utility. We demonstrate the utility of our approach by giving a methodology for examining sequences of function calls and showing experiments that result in manageable amounts of understandable data about program executions. In most of the instances that we present, our techniques offer an improvement over existing techniques.

3.2 Methods

With post mortem analysis, a system can record more data, and analysts can examine the data more thoroughly than in real time intrusion detection. Ideally, the analysts have available a complete record of execution, which enables them to classify sequences as "rare" or "absent." A real-time intrusion detection system, on the other hand, must classify sequences without a complete record because not all executions have terminated. Hence, if a sequence generally occurs near the end of an execution, classifying them as "absent" in the beginning would produce a false positive.

Our anomaly detection techniques use an *instance-based machine learning* method previously used for anomaly detection using sequences of system calls [FHSL96, HFS99], because it is simple to implement and comparable in effectiveness to the other methods. However, rather than system calls, as were previously used, we do so using function calls and sometimes also indications of the points at which functions return (hereafter "returns"). Generally, instance-based machine learning compares new instances of data, whose class is unknown, with existing instances whose class is known. In this case, the experimenters compared windows of system calls of a specific size between test data and data known to be non-anomalous, using *Hamming distances.* The original research was done over a number of years, and the definition of *anomaly* changed over time. At some points, an anomaly was flagged when any Hamming distance greater than zero appeared. At other times, an anomaly was flagged when Hamming distances were large, or when many mismatches occurred. An optimal window size was not determined. The window size of six used in the experiments was shown to be an artifact of the data used, and not a generally recommended one [TM02].

Currently, our instance-based learning uses a script to separate the calls into sequences of length k, with k from the set 1..20. We calculate the Hamming distance between all "safe" sequences and the new sequences for several different values of k.

Sequence length is important for anomaly detection. However, the length to consider depends on a number of factors. If k is small, the sequences may not be long enough for an analyst to separate normal and anomalous sequences. Also, short sequences can be so common that they may be in the "safe" corpus even if they are part of an anomalous sequence at another time. On the other hand, with instance-based learning techniques, the number of distinct sequences of length k increases exponentially as k increases linearly. Also, the number of anomalous sequences that a human analyst has to look at grows as well, though not exponentially. Through experimentation, we discovered that values of k larger than 10 generally should be avoided.

Generally, since our analysis is post mortem, we are more concerned about the human analyst's efficiency and effectiveness than with computing efficiency. By using automated, parallel pre-processing that presents options for several values of k, a forensic analyst can decide which sequence lengths to use. We show the effects of choosing different values of k in this dissertation, but do not claim that a particular value of k is ideal. Ultimately, given that forensic analysis will remain a lengthy, iterative process, the sequence length parameter is one that a human analyst will choose and vary according to the situation being analyzed.

In addition to function *calls*, it is also sometimes useful in finding anomalies in the sequences to use function *returns*, as we describe in our experiments. For this reason, we collect both calls and returns in our implementation, but only actually use the returns when necessary.

Anomaly detection is the foundation for our forensic analysis. The anomaly detection process flags anomalous executions, and presents the relevant data for further study. Whereas the anomaly detection process is automated, the forensic process involves a human. Though forensic analysis will undoubtedly be more easily automated in the future, automation is currently hard.

To begin the forensic analysis, a human analyst first decides which sequence length to choose to investigate further. Given that we calculate the number of differing sequences for several values of k, the analyst should choose one that is manageable to look at, but one in which anomalies are present.

3.3 Experiments and Results

We compared sequences of function calls from an original (non-anomalous) program with several versions of that same program modified to violate some security policy. Our goal was to determine how readily the differences could be detected and what we could learn about them. We chose the experiments as examples of important and common classes of exploits, as identified and enumerated in the seminal RISOS [ACD+76] and Protecton Analysis (PA) [BH78] reports.

We ran the first four experiments on an Intel-based, uniprocessor machine running FreeBSD 5.4. In those experiments, we began by using Intel's dynamic instrumentation tool *Pin*

[LCM+05] to instrument the original, and modified versions of the programs to record all function calls made. In the last experiment, we used the `ltrace` tool on an Intel-based, uniprocessor machine running Fedora Core 4. The `ltrace` tool captures only dynamic library calls, rather than user function calls, but unlike the *Pin* tool, is type-aware, and therefore enables analysis of parameters and return values. System calls are captured by both instrumentation methods.[2]

To create a database of calls to test against, we ran unmodified versions of the binaries one or more times. For example, for our experiments with `su` below, one of the variations that we tested included successful as well as unsuccessful login attempts.

In the experiments, some sequences appeared multiple times in a program's execution. We refer to the number of *distinct sequences* in an execution, counting multiple occurrences only once. (The *total number* of sequences is simply the total number of calls, minus the length of the sequence, k, plus 1.) When we compare the safe program's execution to the modified program's execution, we refer to the sequences appearing only in one execution, and not the other, as the *different sequences*. The relevant numbers are the number of *total different sequences* in each version, and the number of *distinct different sequences* in each version, where again, multiple occurrences are counted only once.

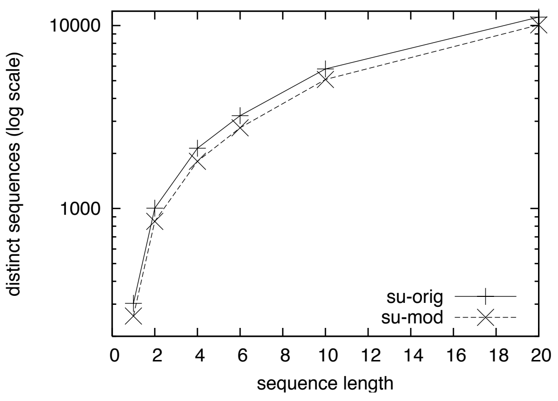
Omitting and Ignoring Authentication

su Experiment #1. Our first experiment illustrates a simple, manually-constructed anomaly. We compared the execution of a normal, unaltered version of the UNIX `su` utility with one in which the call to `pam_authenticate` was removed, thus removing the need for a user to authenticate when using su.

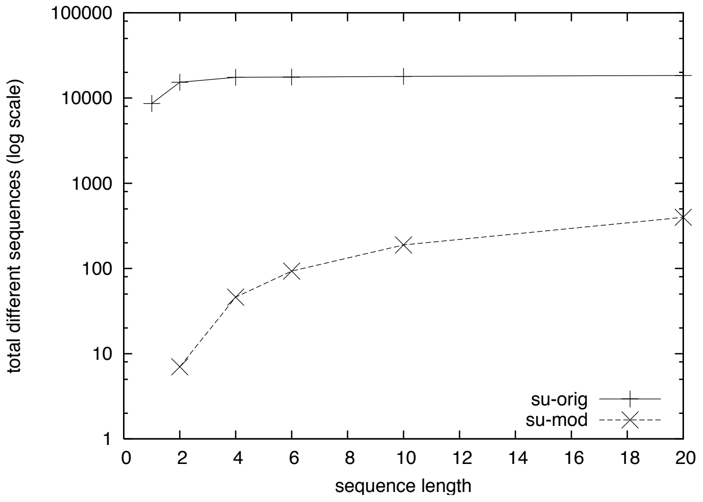
Figure 3.1(a) shows the number of distinct sequences of function calls for executions of the two `su` programs. Figure 3.1(b) shows the number of sequences of function calls each appearing only in one version of the execution but not the other. The `su`-mod curve in Figure 3.1(b) quickly jumps from 7 when $k = 2$, to 46 when $k = 4$, and 93 when $k = 6$. This pattern is typical and emphasizes why larger values of k can be a hinderance to understanding the data. 93 sequences of 6 is a lot of data to analyze visually. Although shorter sequences may fail to highlight intrusions, longer sequences can present overwhelming amounts of data to a forensic analyst.

Choosing $k = 4$ somewhat arbitrarily, here is the sequence information for the original and modified versions of `su`:

[2]We capture both the system call and `libc` interface to the system call, so we can determine when a function call to `libc` just calls a system call and when it does not.



(a) number of distinct function call sequences in original and modified



(b) function call sequences present only in a single version

Figure 3.1: Unique and different numbers of function call sequences in the original version of su, and the version with `pam_authenticate` removed.

$k = 4$	number of sequences in each execution
su-original	37142 (2136 distinct)
su-modified	8630 (1812 distinct)

The discrepancy between the number of sequences in each version is sufficient to determine that *something* is different between the two executions, but little more can be learned. Therefore we compared the data between the two more directly. Figure 3.1(b) shows plots of the number of calls appearing only in one version of su and not in the other (using a logarithmic scale). Again, as k grows, the number of sequences appearing in the original version and not in the modified one becomes quickly too large for a human to analyze easily (unless obvious patterns are present), but the number of sequences appearing in the modified version and not the original stays easily viewable until a sequence length of about 6 is used. We chose a length of 4. Using the instance-based method described earlier, a comparison between sequences in the original and modified versions of su shows:

$k = 4$	different sequences
appearing only in su-original	17497 (370 distinct)
appearing only in su-modified	46 (all 46 distinct)

Of the 370 distinct sequences appearing only in the original version, we also learned that 14 sequences of length 4 were called with unusually high frequency:

$k = 4$ sequences	# total occurrences	% of total program
MD5Update,memcpy,MD5Update,memcpy	3533	9.51%
memcpy,MD5Update,memcpy,MD5Update	1528	4.11%
MD5Final,MD5Pad,MD5Update,memcpy	1002	2.70%
memcpy,MD5Update,memcpy,memcpy	1002	2.70%
memcpy,memcpy,memcpy,memcpy	1002	2.70%
MD5Update,memcpy,memcpy,memcpy	1002	2.70%
MD5Init,MD5Update,memcpy,MD5Update	1002	2.70%
MD5Pad,MD5Update,memcpy,MD5Update	1002	2.70%
memcpy,MD5Update,memcpy,MD5Final	1001	2.70%
memcpy,MD5Final,MD5Pad,MD5Update	1001	2.70%
MD5Update,memcpy,MD5Final,MD5Pad	1001	2.70%
memcpy,memcpy,memset,MD5Init	1000	2.69%
memcpy,memcpy,MD5Init,MD5Update	1000	2.69%
memcpy,MD5Init,MD5Update,memcpy	1000	2.69%
Total	17076	46.0%

These 14 sequences represent nearly half the entire program execution. Looking more closely at the 14 sequences (which is easy to do, since they stand out so voluminously from the others), all but one are clearly related to MD5, which does checksumming and encryption. This is an obvious clue to a forensic analyst that authentication is involved in the anomaly.

By comparison, it would not have been obvious what functionality was removed had we looked only at kernel calls, because they tend to be more utilitarian and less descriptive. For example, setting $k = 1$ (note that a sequence of length $k = 1$ is the same thing as a single call) and looking only at kernel calls, the calls absent in the modified version of `su`, but present in the original, were `setitimer` and `write`. Given $k = 2$, the number of sequences present in one and not the other jumps considerably. Among the sequences were the calls `fstat`, `ioctl`, `nosys`, `sigaction`, `getuid`, and `stat`. These are clearly useful pieces of information, but not as descriptive as function calls.

Might there be an even easier way of finding the anomalous function call sequences? When $k = 2$, the results change significantly:

$k = 2$	different distinct sequences
appearing only in `su`-original	161
appearing only in `su`-modified	7

The reduced number of anomalous sequences makes the data much easier to look through. Using $k = 1$:

$k = 1$	different distinct sequences
appearing only in `su`-original	45
appearing only in `su`-modified	0

In fact, we can summarize the relevant calls for $k = 1$ in four lines:

$k = 1$ sequence	# total occurrences	% of total program
`MD5Update`	5538	14.91%
`MD5Final`	1002	2.70%
`MD5Init`	1002	2.70%
`MD5Pad`	1002	2.70%
Total	9547	18.69%

In this experiment, $k = 1$ provides a list of differing function calls that provided large clues about what was anomalous in the program. This is not always the case. In experiments with other programs, we discovered $k = 1$ showed no differences at all and a minimum value of $k = 2$ or even $k = 4$ was needed. Likewise, we discovered that $k = 4$ provided manageable results similar to those in this experiment, but $k > 4$ provided too many. In describing future experiments, we will choose a value of k that shows a differing number of sequences for at least one of the code versions to be greater than 1 and less than 20. In most cases, that means either $k = 2$ or $k = 4$.

The experiment also showed that function calls provide additional value beyond that provided by system calls. Using $k = 2$ and looking only at system calls, there are 23 sequences of system calls (19 distinct) occurring in the original version of `su`, and not the modified. The most significant differences are as follows:

$k = 2$ sequences of system calls only in su-orig	# total occurrences
`ioctl, write`	2 (0.27%)
`nosys, fstat`	2 (0.27%)
`getuid, stat`	2 (0.27%)
`open, close`	2 (0.27%)
`lstat, open`	2 (0.27%)

These sequences suggest that an anomaly is occurring, but do not describe what the anomaly is. Indeed, none of the sequences would provide any indication to most forensic analysts as to where to look in the source for the anomalous behavior. Contrast this to the much more useful perspective that the sequences of function calls provided. Also, we can see that though the amount of data captured by recording function calls rather than system calls alone is 20–200 times higher, the amount of data necessary to for an analyst to examine is not nearly as high. In this case, the number of distinct, different function call sequences is only 7 times higher than the number of distinct, different system call sequences, with some sequences appearing so frequently that they immediately stand out. The function call data is more useful and does not require much more work to examine.

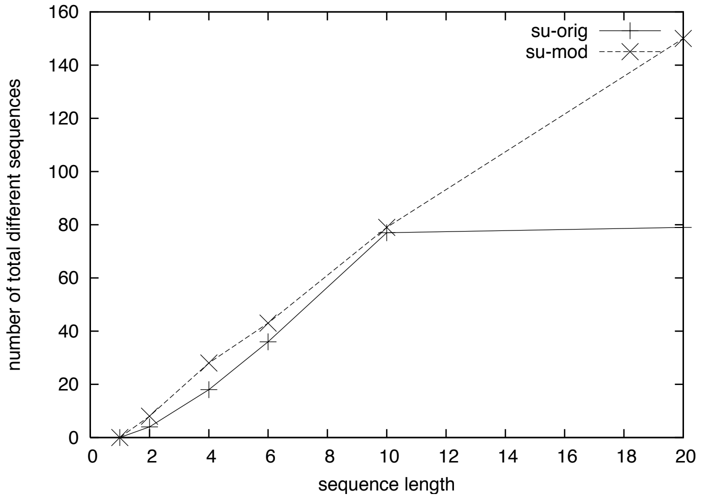


Figure 3.2: Number of function call sequences appearing only in the original version of su, and the version modified to ignore the results of pam_authenticate.

su Experiment #2. We performed a second experiment with su, where we modified su to run pam_authenticate, but ignore the results rather than just removing the function entirely. In Figure 3.2, we show a number of sequences appearing in one version, but not the other. Again, we see that a sequence of length 4 gives a manageable amount of results for sequences appearing only in the original version of su, with 4 of the 13 sequences being:

$k = 4$ sequences in `su`-original, not in `su`-modified
`strcmp,pam_set_item,memset,free`
`crypt_to64,crypt_to64,strcmp,pam_set_item`
`crypt_to64,strcmp,pam_set_item,memset`
`sys_wait4,login_getcapnum,cgetstr,cgetcap)`

That said, $k = 2$ still gives us all we need to investigate the anomaly. For a sequence of length 2, there are 13 distinct sequences occurring in `su-original` and not in `su-modified`. One is "`strcmp,pam_set_item`," which is sufficient to raise concerns in any forensic analyst's mind because it indicates that the result of the authentication is not being set (`pam_set_item`) after the check (`strcmp`).

By comparison, looking only at system call traces, results are again less forensically useful because they are less descriptive. There are no different sequences of system calls with $k = 1$ or $k = 2$. With $k = 4$, we see 3 system calls (all 3 distinct) in the original version and not in the modified and 4 system calls (all 4 distinct) in the modified version and not in the original. Unlike function calls indicating a relationship with authentication and cryptographic routines, however, we instead see:

syscall sequences only in `su`-modified	syscall sequences only in `su`-original
`ioctl,close,close,sigaction`	`ioctl,close,sigaction,close`
`setpgid,ioctl,close,close`	`sigaction,close,sigaction,sigaction`
`close,close,sigaction,sigaction`	`setpgid,ioctl,close,sigaction`
	`close,sigaction,close,sigaction`

The above table indicates something suspicious involving a socket call, leading to an inference that there is a problem involving interprocess communication. There is nothing indicating the nature of the problem. This again emphasizes the value of function call sequences, which clearly show an authentication issue. In what follows, we shall focus only on function call sequences rather than the differences between function call sequences and system call sequences.

Spyware

ssh Experiment #1. `ssh` is key to accessing systems over a network. Hence, it offers opportunities for malice, especially when the attacker has access to a password, private key, or other authentication token. Consider two versions of the `ssh` client: the original, and one that is modified to do nothing more than echo the password back to the terminal. We wish to determine which sequence length will alert the analyst to the change. Figure 3.3 shows the number of sequences that exist only in the executions of the original and modified versions of the `ssh` client. When a human analyst looks at a list of function call sequences flagged as anomalous, she can most easily spot differences when at least one list of sequences appearing in one execution but not the other is relatively short. In this case, sequence lengths of k larger than 2 or 4 can cause the resulting sequences to become unmanageable. To the degree that an anomaly can be caught by sequences of 2 or 4, an analyst should use those smaller values. It is difficult for an analyst to determine whether she has captured an anomaly with a sequence length of 2 or 4, but if an anomaly exists at all, a larger value of k is probably unnecessary.

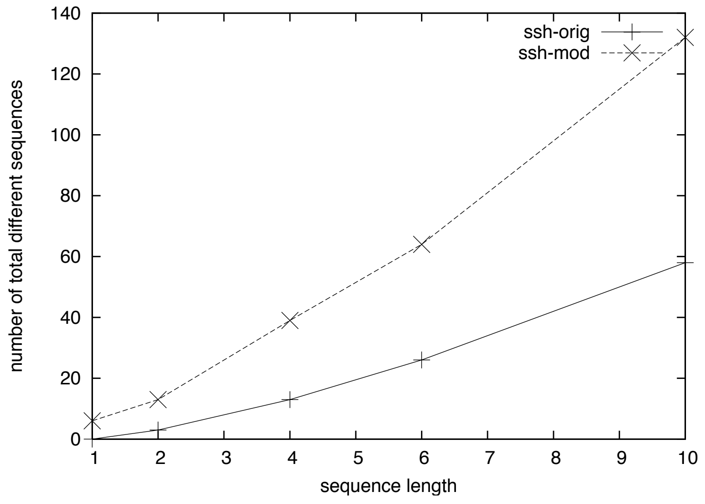


Figure 3.3: Number of function call sequences appearing only in the original version of `ssh`, and the version modified to echo the password back.

In this case, the comparison between each version, with sequence length $k = 4$, results in the following:

$k = 4$	total different sequences
appearing only in `ssh`-original	13 (13 distinct)
appearing only in `ssh`-modified	39 (39 distinct)

Many of the functions in the anomalous sequences have obscure names: `rlock_release`, `rtld_bind`, `rtld_bind_start`, and `localeconv`. Although the sequences clearly differ in the executions of both programs, the interpretation of the differences is not clear. Therefore, we include the function *returns* as tokens, in addition to function call points. In what follows, the function returns are indicated with the suffix `-RET`.

$k = 2$	total different sequences, including function returns
appearing only in `ssh`-original	6 (3 distinct)
appearing only in `ssh`-modified	25 (25 distinct)

Now, using the additional data along with our previous methods, we discover the following variances in between all of the somewhat obscure sequences. In them, one function stands out immediately: `read_passphrase`. The following table shows sample sequences that appear only in each version of `ssh`:

$k = 4$ sequences in `ssh-original`, including function returns
`memset-RET,read_passphrase-RET,input_userauth_info_req,packet_put_cstring`
`read_passphrase,memset-RET,read_passphrase-RET,input_userauth_info_req`
`xstrdup-RET,read_passphrase,memset-RET,read_passphrase-RET`

$k = 4$ sequences in `ssh-modified`, including function returns
`fprintf-RET,read_passphrase-RET,input_userauth_info_req,packet_put_cstring`
`vfprintf-RET,fprintf-RET,read_passphrase-RET,input_userauth_info_req`
`memset-RET,read_passphrase,rtld_bind_start,rtld_bind`

These sequences indicate a location to examine more closely. Knowing about the `read_passphrase` call directs an analyst's attention to the routine in the source code, simplifying the investigation.

We mentioned earlier that shorter sequences would potentially be even easier to analyze, if they catch the anomaly. As it turns out, $k = 2$ does catch the anomaly. In the two distinct sequences in the original version, the `read_passphrase` call stands out. In fact, `read_passphrase` is one of only three non-kernel calls in the list, which further indicates what we get by looking at all calls rather than just kernel calls. In the 25 distinct sequences in the modified version, we show a selection of the sequences in the following chart. All but 3 involve string operations, which would help a forensic analyst find the offending location in the source code:

$k = 2$ sequences in `ssh-original`	$k = 2$ sequences in `ssh-modified`
`memset-RET,read_passphrase-RET`	`fprintf-RET,read_passphrase-RET`
`BN_copy-RET,BN_div-RET`	`vfprintf-RET,fprintf-RET`
`BN_div,BN_copy`	`vfprintf,vfprintf`

The inclusion of function returns effectively doubles the amount of data collected. However, as with the addition of function calls to system calls, the increase in amount of data collected does not necessarily result in a proportional increase in the amount of data a human needs to examine. In our experiments, when the amount of data collected doubled, the increase in the amount of data presented to the analyst was often closer to 50%.

Nevertheless, the resulting increase in data to collect and analyze was the reason why we do not use function returns in all of our experiments. There are some cases where function returns are helpful and others where they are unnecessary. As with the value of the sequence length parameter, an analyst will need to decide whether or not function returns should be included.

ssh Experiment #2. This example re-creates a malicious program that was used by an intruder at the San Diego Supercomputer Center in 2004 and 2005 [MB05, Sin05]. In the program, the intruder modified the `ssh` client program to capture user passwords and send the

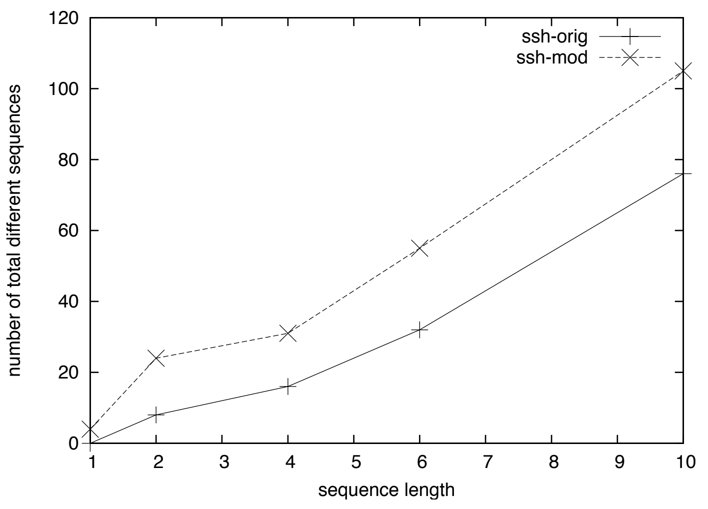


Figure 3.4: Difference in number of function call sequences in original version of `ssh`, and the version modified to send the captured password over a network socket.

passwords via a network socket to another machine, which collected them. The initial comparison shows the following:

$k = 2$	total different sequences
appearing only in `ssh`-original	8 (all 4 distinct)
appearing only in `ssh`-modified	24 (all 12 distinct)

Several calls do stand out: `read_passphrase`, `inet_aton`, `inet_addr`, `socket`, and `sendto`.

$k = 2$ key sequences in `ssh-modified`
`inet_aton-RET,inet_addr-RET`
`read_passphrase,sys_close-RET`
`sys_close-RET,read_passphrase-RET`
`inet_addr,inet_aton`
`inet_addr-RET,read_passphrase`
`socket-RET,read_passphrase`
`sendto-RET,read_passphrase`

In these examples, the four network-related functions, `inet_aton`, `inet_addr`, `socket`, and `sendto` (which is used to send data), might capture an analyst's attention. The first two are `libc` calls, and the second two are system calls. While anomaly detection could have indicated a potential anomaly in the program, based upon the system calls, these forensic techniques draw an analyst's attention to the right place in the code to *understand* the anomaly. In this case, the `libc` calls help flag the anomaly, and using that information, further exploration in the code by a forensic analyst would reveal the purpose of the calls.

One complication results from the way Pin collects function calls and function returns. Due to technical limitations, Pin occasionally misses less than 1% of calls made. For example, note that in the above set of key sequences, `sendto-RET` appears (showing the return from the `sendto` function) but the `sendto` call itself does not appear. Given the low volume of these missed calls, the results of the analysis are not affected.

Ignoring Permissions in the Filesystem In this example, we have re-created an experiment performed previously using anomaly detection techniques [FHSL96, HFS99], but in our experiment have used function calls instead of just system calls. The experiment involves a bug in an older version of `lpr` that allows an attacker to create or overwrite any file on the system [8LG]. In this case, the bug was exploited to replace `/etc/passwd` with `/tmp/passwd`.

Using $k = 2$, since that has been successful so far, we see 48 sequences (38 distinct) in the "test" version of `lpr` that are not in the "safe" corpus. In the following table, the function calls that are part of `lpr` are in boldface font, and the system calls and `libc` calls are in monospaced font:

selected $k = 2$ sequences only in `lpr-modified`	# occurrences
`close, open`	1
`close, close`	2
`seteuid,creat`	1
`seteuid,sys_read`	2
`swhatbuf,sys_fstat`	1
`sys_write,`**nfile**	1
copy,card	1
nfile,`sys_umask`	1
`swrite,swrite`	1
`swrite,rtld_bind_start`	1
`printf,rtld_bind_start`	1
`swsetup,smakebuf`	1
`open,`**copy**	1
`link,error_unthreaded`	1
`sys_unlink,error_unthreaded`	2
`sys_unlink,sys_unlink`	2
`sys_unlink,rtld_bind_start`	1
`vfprintf,vfprintf`	1
`smakebuf,swhatbuf`	1
`ioctl,sfvwrite`	1
`sys_umask,fchown`	1
`sys_read,sys_write`	2
cleanup,`signal`	1
`malloc,isatty`	1
`sys_sigaction,seteuid`	1
`error_unthreaded,sys_unlink`	2

When one is printing a file, one may be surprised to find that files are created, copied, removed, and have their permissions changed in the process. Though each of the sequences in the table above appear only once or twice, we can see repeated calls to sys_unlink and seteuid, which stand out among these results as being important and suspicious. What actually helps to find the bug, however, is the copy function call in line 7 of the table, which is a routine in lpr.c that creates a file and copies a file descriptor.

The copy function contains a call to creat. creat is a legitimate system call, though it it is now deprecated in FreeBSD. An analyst who knows of the deprecation might be suspicious of a call to creat, because unlike open, it is now known to be one of two steps in a non-atomic method for creating and opening a file, which must be used carefully to avoid race conditions. At the time the exploit was released, an analyst would not necessarily have known that creat might be a problem, however.

The sys_unlink and seteuid calls that we see in the table are not themselves the vulnerability, but are the effect of the exploit, since the creat call allows files to be arbitrarily overwritten (sys_unlink) and permissions reset (seteuid).

Buffer Overflows In the first three experiments, we have shown full, real examples of actual exploits for the purpose of demonstrating the complete process of using our methods. However, the results are also partially demonstrating the skill of the analyst, not the simple results of our method. Therefore, in the following two experiments, we use simplified examples rather than actual attacks. This is partially because the simple examples show the essence of the attack and analysis, and embedding them in more complex exploits simply requires that the analyst winnow them out. It is also due to space limitations, as there is not enough room for a detailed analysis of two more large attacks that use these exploits. Therefore, we present examples of the two attack approaches without the extraneous entries.

There are many examples of attacks that exploit buffer overflow vulnerabilities. One can invoke a root shell directly, copy a setuid root version of a shell into /tmp, or, instead of working directly to gain arbitrary access, one can write *shellcode*[3] to perform exactly the desired operations, directly. Unfortunately, at this point, we have not discovered a universal pattern to buffer overflow exploits, and therefore cannot detect one directly. What we can do at this point is understand when the order of events is unusual. Consider the following piece of code [fid01]:

```
int main(int argc, char *argv[]) {
        char buffer[500];
        strcpy(buffer, argv[1]);
        printf("Safe program?");
        return 0;
```

[3]The assembly language instructions in a buffer overflow exploit designed to provide a shell.

```
}
```

We run this program, called `vulnerable`, in two different ways. The first way is one that is "safe" and does not exploit the vulnerability by overflowing the buffer. It simply passes 499 instances of the ASCII character 'a' as the argument to `vulnerable`. The second way exploits the vulnerability. It copies more than 500 characters into the buffer. The characters include *shellcode* that returns a shell owned by `root`.

Although we see a discrepancy between the sequences for the safe and exploited programs, the reason for the difference, at first, is unclear. As we did in a previous example, we add indications of where the functions return. The following variances appear:

$k = 4$ sequences only in non-overflowed `vulnerable`
`vfprintf-RE, printf-RET, main-RET, start`
`printf-RET, main-RET, start, rtld_bind_start`
`main-RET, start, rtld_bind_start, rtld_bind`

The exploited program contains a sequence of length 3: `vprintf-RET`, `printf-RET`, `main-RET`. The non-exploited version of the program continues after `main` ends with additional functions, and finally, ends with the `exit` syscall. The exploited version stops at the end of `main`. A forensic analyst who saw these results would realize that `exit` is being skipped in one version (the exploited version), most likely because the program has been altered. This means that all the cleanup functions `exit` invokes are also skipped.

Our difficulty in analyzing buffer overflows stems from a limit of the dynamic instrumentation tool we used. Pin currently allows analysts to instrument code written to the stack and then executed, but a special command must be called between the time the code is written and the time it is executed. Unfortunately, implementing this special command would distort our view of the buffer overflow. That said, the evidence of the buffer overflow already exists in our view, even if the exact nature of the overflow does not. The effects of some exploits will be more apparent than the exploit itself. In this case, the exploited program does not exit (the effect) because it is actually spawning a program (the exploit itself, which we cannot see but can infer).

Trojan Horses In an early version of UNIX, there was a flaw in `vi`[4] such that when an editing session was terminated unexpectedly, it would mail the user about how to recover data from the lost session, by calling the mail program without a full path (e.g. `/bin/mail`, but with simply a program name: `mail`). As a result, if a binary called `mail` were placed earlier in the search

[4]See p.90, slide #171 in *How Attackers Break Programs, and How to Write Programs More Securely* [Bis02].

path order than /bin, the binary, which could be a Trojan horse, would be executed. Further, since vi used a privileged program to store temporary files on the system, and that program called mail, the Trojaned version of mail would be executed with superuser privileges.

Our experiment involves a very minimal program:

```
int main() {
        FILE * F;
        F = popen("date","r");
        printf("Opened\n");
        pclose(F);
}
```

In this program, the date program is called with popen, just as the original bug in vi. The circumstances are such that the path is set to:

```
.:/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin
```

The system's date program is located in /bin. In the first case, that is the only date in the path. In the second case, a "rogue" date program is also in the current working directory. In reality, it could be easier to modify a user's path without their knowledge, or to put a malicious program higher in their "search path" than the system's, than to obtain root access. This is because inserting a malicious program only requires access to a *user's* account, not root. Therefore, any number of password-sniffing, social engineering, or other techniques might enable an attacker to do this.

We test the same program under both circumstances. Differences in each case show up with values of $k >= 6$. With $k = 6$, and comparing the library calls of the program that is being "tricked" into executing the Trojan, not the Trojan itself, the results show the following:

$k = 6$ sequences only in version calling Trojan
malloc,strncpy,xstat64,malloc,xstat64,access
strncpy,xstat64,malloc,xstat64,access,free
xstat64,malloc,xstat64,access,free,xstat64

Without even looking at the function parameters themselves, in the first execution, we can see multiple calls relating to malloc, strncpy, access, free, and xstat64.

Looking more carefully at the raw data of function calls, parameters, and return values (the latter two are available as a result of using ltrace in this experiment rather than *Pin*), we can see why: the version that runs the system date does an xstat64 (related to stat) on the current working directory and /sbin before /bin. Further, the first two return non-zero values, which indicates the non-existence of a file, while the `xstat64` returns zero in the `/bin` directory for `date`. Conversely, when the Trojaned `date` is in the directory, there is only one `xstat64`, and the program goes no further.

Intuitively, we know why this is the case. The version of the test program that gets tricked into calling the Trojan searches its path (and hence uses `xstat64`) in different sequences than the version that calls the system `date`. The results are misleading, given that there are actually *more* total sequences in the version calling the system `date` than the Trojan, but more *distinct* sequences in the other version. This is an anomaly that sometimes occurs. In this case, it occurs with $k < 14$. Nevertheless, the experiment shows how sequences of function calls are helpful in analyzing Trojan horses.

The results of this experiment show not only the value of tracing library calls across forks, but also the value of and ability to analyze call parameters and return values.

3.4 Conclusions on Forensics Using Sequences of Function Calls

Despite the overlap between the the intrusion detection and forensic analysis processes, they are distinct problems. Because of this, there has not been as much cross-fertilization as one might wish. In this chapter, we have borrowed one technique from intrusion detection and, with modifications, successfully applied it to forensic analysis.

We believe that there are a number of reasons why this technique has been successful. For example, by collecting function calls at runtime and within user space, we gain significantly more useful information than if we collect system calls alone. Similarly, our technique would not be as successful if we were to look only at single calls (that is, $k = 1$). The fact that this approach looks at *sequences* of function calls, and therefore largely addresses both cause and effect, significantly aids in its effectiveness. Finally, one of the key benefits of applying intrusion detection techniques to forensics is to aid an analyst in understanding the information by highlighting the suspicious parts of the execution. This feature significantly aids a forensic analyst by reducing the amount of data that the analyst needs to look at in order to analyze an attack.

A variety of techniques could help improve these methods, including analysis using Hidden Markov Models (HMMs) and data mining. We would also like to analyze not only anomalous programs but also anomalous sessions and users. Tools that help to make logs and logging mechanisms more tamper-proof [GFM$^+$05, Lym04, SK99, ZMAB03] would also be useful.

The set of all orderings of safe events on a computer system is finite (as a computer is a finite state machine) and, upon many repeated executions, the total number of distinct sequences actually executed levels off, asymptotically. If an ordering of events in that set occurs, the program is behaving anomalously. We can not always determine this in real-time, however, because non-deterministic events, such as user inputs and variable system state, cause programs to perform deterministic events at different times. But, when known, "good" events do happen, they do so in an order that can often be predicted by gathering enough data about the safe operation of a program ahead of time. Therefore, we need to use anomaly detection techniques on data in a *post mortem* manner, even though this only tells an analyst whether the program is anomalous after the data for the entire program has been analyzed.

Anomalies among short sequences of function calls indicate not only that a *process* is anomalous but also the *part* of the process (code) that is anomalous. This significantly reduces the amount of the recorded function call data that actually needs to be analyzed by a human. We can then check the source code. A fundamental requirement of forensics is to understand not only *if* an anomaly is occurring, but also exactly *where* it occurs. This requires detail without overwhelming volumes of data. Anomalous sequences of function calls fit this criteria better than other levels of abstraction and analysis methods previously used. Function calls are also a helpful abstraction, because humans generally find them more descriptive than system calls alone. While we do not claim that function calls, rather than system calls alone, are *required* to analyze all attacks, we have shown that function calls can help to analyze the attacks much more easily due to the descriptiveness of the calls. Function call sequence length, previously shown to have no optimal answer, appears to be most desirable when $1 \leq k \leq 10$, because it produces manageable but not overwhelming amounts of data.

That a program execution has a finite set of function call sequences suggests that it should be possible to model programs formally as sequences of function calls. Correlating the types of anomalies that an analyst looks for generates automated, common models of anomalous behavior based on those models. At present, we do not know enough about the behavior of most programs to formulate such a formal specification, or some other formal model that would define the vulnerabilities better [Bis99]. Given the success of our experiments, the development of more universal analysis techniques and formal models that show commonalities between exploits seems promising.

Anomaly detection techniques on function call data in post-mortem analysis allows one to find unexpected events, including the absence of expected function calls as well as the presence of unexpected function calls. This enables an analyst to discover when an event that should occur does not.

Some techniques fail. Looking for calls that are "rare" in one version and "common" in the other did not bear much fruit in our experiments, although even those failed experiments suggest that the technique has potential value, and further experimentation is warranted. Additional, more descriptive information, should help address previously unanalyzable scenarios. However, as we noted earlier, evaluation of these techniques is difficult, and ultimately will require further study to determine which technique is most effective.

Our experience with developing this method, as well as analysis of other existing approaches has suggested to us a number of generalizations — principles — that we believe should guide future solutions to forensic analysis in order to be most effective. We discuss those principles in the next chapter.

The work presented in this chapter was first described in an earlier paper, "Analysis of Computer Intrusions Using Sequences of Function Calls," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, conditionally accepted with minor revisions by *IEEE Transactions on Dependable and Secure Computing (TDSC)*, January 2007.

4

Toward Forensic Models

Sherlock Holmes: *"It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."*

—Sir Arthur Conan Doyle, "A Scandal in Bohemia," *The Strand Magazine* (1891)

The work presented in this chapter was first described in two earlier papers by Peisert, et al. [PBKM05, PBKM07b].

Our research in performing forensic analysis sequences of function calls, presented in the previous chapter, was one response to our perception of shortcomings in existing forensic solutions. This chapter presents a more general overview of some of the high-level issues in computer forensics and a set of principles to help overcome the shortcomings in existing solutions. Finally, it discusses a set of qualities and guidelines, adapted from the principles, that we believe a good forensic model should possess and adhere to.

4.1 Principles of Forensic Analysis

Successful forensic analysis requires the ability to re-create any event regardless of the intent of the user, the nature of the previous events, and whether the cause of the events was an illegitimate intruder or an authorized insider. The ability to do this has progressed very little since the first research on the subject in 1980 [And80]. Examples of events that current tools cannot accurately identify include anything that a program is reading or writing to memory. This covers a huge number of possible events and exploits, such as changes to the user environment, covert channels, buffer overflows, and race conditions.

Current techniques to resolve these problems result in generation of too much information, leading to impractical performance slowdowns and high storage requirements. They also suffer from a disparity between the goals of system designers, administrators, and forensic analysts.

Based on our analyses of the shortcomings of existing tools, we outline five principles that address many of the current failures. Failing to follow all of the principles presented here may cause a tool to fail to record actions, or results, in enough detail to understand their meaning. Several of these principles parallel principles from classic operating system concepts, such as having fail-safe defaults.

No widely-used operating system could, in practice, record every event, its cause, and its result. Hence we need other ways to generate the information needed for forensic analysis. Three key bases for computer forensics will help to interpret events correctly:

- *The entire system must be considered.*
- *The effects of an action can be significantly different than what we expected them to be.*
- *Runtime data is the only authoritative record of what happened. While pre-intrusion static vulnerability scans, and post-intrusion analyses of system state can often be of enormous help, a complete set of runtime data is the only authoritative set of data that can be relied upon for forensic analysis in all circumstances.*

For modern operating systems, these ideas suggest the following principles. In these, we use *context* as a broad term that includes any system detail surrounding an event. *Environments* are a subset of *context* and refer generally to user-definable shell and program settings.

Principle 1: *Consider the entire system. This includes the user space as well as the* entire *kernel space, filesystem, network stack, and other related subsystems.*

Principle 2: *Log information without regard to assumptions about expected failures, attacks, and attackers. Trust no user and trust no policy, as an analyst may not know what is needed for a particular attack in advance.*

Principle 3: *Consider the effects of events, not just the actions that caused them, and how those effects may be altered by context and environment.*

Principle 4: *Consider context to assist in interpreting and understanding the meaning of an event.*

Principle 5: *Present and process every action and every result in a way that can be analyzed and understood by a human forensic analyst.*

Examples of problems[1] that current tools do not address, and the principles that would need to be followed to address them, include:

1. changes in the *user enviroment* (such as the UNIX shell or applications) (principles 1, 3, and 4)
2. changes in the *global system enviroment* (such as file permissions) that affects system operation at a later time (principles 1, 3, and 4)
3. *abstraction shortcuts* bypassing standard mechanisms, such as bypassing the filesystem to read or write to the raw disk (principles 1 and 4)
4. buffer and numeric *overflows* and *reading memory* locations larger than the allocated buffer (principles 1 and 4)
5. *race conditions* (principles 1, 2, and 4)
6. *programmer backdoors* exploited (principles 1, 2, and 4)
7. *code injected* into the program instruction stream (principles 1, 2, and 4)
8. *code written to the heap* at runtime and executed dynamically (principles 1, 2, 3, and 4)
9. *self-modifying code* whose behavior varies based on environmental conditions (principles 1, 2, 3, and 4)
10. *illicit channels* with high-volume data rates, such as data exposed from one user space to another with a memory write or data read from or written to raw disk at points with unallocated inodes (principles 1 and 4)
11. interception of *user input* (principles 1 and 2)

prydain **example** For the rest of this chapter, we refer to a real system compromise as an example. This elements of the compromise are highly representative of elements of most common intrusions. A Mandrake Linux system, called *prydain* was running a wide variety of security software, including *syslog*, *TCPWrappers* [Ven92], the network IDS *snort* [Roe99], the host-level firewall *iptables*, and *Tripwire* [KS94]. All current security patches had been applied. Despite these typical precautions, the machine was compromised. This was discovered from email from a system administrator at another site, whose machines were being attacked by the compromised system. The vulnerability was probably in Linux-PAM (Linux Pluggable Authentication Module), though that suspicion is an inference from available and missing data. No hard evidence is available. The only evidence unearthed by standard procedures and tools was a directory containing a tool to perform brute-force `ssh` attempts against other machines, `ctime` evidence that

[1]This list is based on work by Gilliam, et al. [GWSB03], which in turn was adapted from a list of the top 50 software problems identified by NASA [NAS].

a number of standard binaries had been replaced and possibly "trojaned," and syslog messages showing a number of successful `ssh` logins for every user on the system that did *not* have a login shell. No proof of how the intruder broke in and what the intruder did was found.

Next, we discuss the inadequacy of current forensic solutions in terms of our principles.

4.2 Current Problems with Forensics

4.2.1 Principle 1: Consider the Entire System

Principle 1 requires an analyst to have access to at least as much data after an intrusion as an intruder had before and during. For instance, failing to consider user space leaves the analyst with no way to determine whether a buffer overflow occurred (because they miss the write to memory), as in the *prydain* example mentioned above. An intruder can cause a buffer overflow, but using current tools, an analyst cannot prove that one occurred.

Many tools are able to provide forensic information about the kernel space by instrumenting the entrances to the kernel. Many tools attempt containment, or perform static or dynamic analyses for race conditions, buffer overflows, and other potential security exploits. But principle 1 requires that more information be recorded, specifically from both the kernel space and the user space. Even in highly confined systems, an attacker can perform actions relevant to an intrusion in user space. User space environments can alter the perceived nature of an event, and the events reported by a tool that only looks at the kernel space may be inaccurate or incomplete at best.

For instance, current tools identify `mmap` or `sbrk` memory allocation system calls (syscalls), but do not tell us the size and content of data transfered to or from memory. Current tools identify a running process, but do not tell us which function in the code was exploited via buffer overflow to put the program in that state. Current tools can tell us which library was loaded and which file was executed, but not the directories in the user's dynamic library and execution paths that were searched, which would include information on how a user is attempting to explore the machine. Unlike many previous forensic tools and techniques [Bis03, Gro97, ON01, SL03] or even fault tolerant operating systems, today's tools reduce the need for pre-determination of "attack" or "failure" methods by casting a broad net. But they omit non-kernel space data and information about user space that results in missing many forensically-relevant events.

In Chapter 2, we discussed a number of existing tools used for forensic analysis. None of these tools, either collectively or independently, are designed to consider the entire system. Thus, they fail to record the information that analysts require to understand many attacks. Despite their use, both initial knowledge of intrusions and methods of discovery are frequently based on guesses, as in our example above. In that example, we wanted to know, with high certainty, which software contained the vulnerability. What was the nature of that vulnerability? What backdoors were subsequently installed? What actions were taken on the system? What other systems were attacked from our system? How, and were any successful? No current tool is able to answer these questions well because no tools consider the entire system. Even the most basic forensic procedure of halting a machine to make images of the disks has a trade-off between preserving the contents of the disks, and wiping out the content of memory. Hence, all of these tools possess significant shortcomings for forensic analysis of many types of attacks.

However, those tools could be augmented to provide a more complete system viewpoint. Unfortunately, those augmentations suffer from their own incomplete system view. For instance, standard UNIX *process accounting* is trivially bypassed by changing the names of the programs that are run, running a process that does not exit, or using command arguments, which alter the behavior of the program but are not recorded. In the *prydain* example, process accounting would not have recorded the nature of the exploit because the exploit likely hijacked an existing process. Therefore this technique does not address principle 1 either.

Two of the tools previously discussed in Chapter 2, *BSM* and *BackTracker*, gather enhanced data that is useful for forensic analysis. However, they do not consider user space at all, and therefore will not uncover what caused the *prydain* example, nor the exploits that we outlined in Section 4.1.

Similarly, even one of the model-based approaches to forensic anlaysis [Kup04] looks only at BSM data and data gathered from system library interposition. It ignores memory and user space. It ignores a user bypassing dynamic system libraries by using their own static libraries (also violating principle 2). This model-based approach therefore also fails to help analyze the *prydain* example.

ReVirt [DKC$^+$02] is in some senses, the only tool to address principle 1 completely by providing an exact, replayable record of all *non-deterministic* events. This is, in some sense, the "ultimate" logging system. It records the least amount of information needed to re-create everything. But the recording of non-deterministic events is merely the first part of forensics, and is not analysis or auditing. *IntroVirt* [JKDC05] enhances ReVirt by leveraging ReVirt to do *virtual machine introspection.* Hooks in IntroVirt can monitor raw device I/O, CPU exceptions, kernel backdoors, syscall race conditions, file system race conditions, and virtual registers, RAM, and hard disks. At this time, IntroVirt does not record memory events, and program-specific events and environment information, but could be expanded to do so. In that sense, IntroVirt might make an excellent platform for forensic analysis in the future. However, as demonstrated by IntroVirt's current state, *there are many events that occur in an operating system that are irrelevant to forensics, and these events are easier to define and ignore than the events that* are *relevant to forensics.* Starting with this assumption would give analysts a more fail-safe method for having necessary information, by always erring on the side of having *too much* information.

4.2.2 Principle 2: Log Information without Regard to Assumptions

Depending on assumptions about an opponent's abilities can cause an analyst to pay attention to what an attacker wants the analyst to see (a blatant component to an attack), and not the damage an attacker is actually doing (more subtle and concealed by the blatant component). It is difficult enough to gather proof of the results of a non-malicious user's actions, let alone the intentions of a skilled intruder.

For example, assuming that the only threats are from *outsiders* can cause an analyst to ignore *insiders*,[2] therefore placing too much attention on access control mechanisms rather than recording system events. *Any user can be a threat.* One of our key desired outcomes is the ability to detect, and determine with a high level of confidence, when this occurs. In the *prydain* example, all of the secure and encrypted access controls were not enough to stop or even log a remote exploit.

The less data recorded the easier it is for an intruder to distract an analyst. For example, assuming that security violations will only occur in the filesystem can cause an analyst to rely on tools that only look at the filesystem, thereby drawing an analyst's attention to the blatant file accesses. In fact, with root access, the intruder can covertly write those files to another user's memory, and that other user may be the one who actually transports the stolen data off the system.

All of the tools discussed in the previous section fail to record the entire system for one of two basic reasons. Either their designers felt it would be impractical or assumed it was not necessary. In relying on the standard suite of tools, system administrators effectively assume that important events will be logged. In relying on process accounting, administrators effectively assume that binaries are what they appear to be and that arguments to those binaries are irrelevant. In relying on Tripwire, administrators effectively assume that files will show evidence of change if an intrusion has occurred. We have argued in the previous section that

[2] "The *insider* is a user who may take action that would violate some set of rules in the security policy were the user not trusted. The insider is trusted to take the action only when appropriate, as determined by the insider's discretion." [Bis05b]

these assumptions made by forensic tool designers and administrators are not generally safe. Indeed, these tools would not have helped in the *prydain* example.

In the previous section, we mentioned that BSM's audit trails are too coarse to capture and enable reconstruction of many events, because BSM makes assumptions about what is relevant to security. A huge number of events, particularly in user space, are not considered "security-related" events, and therefore BSM does not record them. In the *prydain* example, BSM might have provided additional evidence that could help an analyst infer a cause of the intrusion — for example, by showing a series of process executions and file manipulations. But it would not have provided proof of a buffer overflow exploit, because BSM does not look at user space. Even BackTracker relies on the assumption that an analyst will have evidence in the form of a process ID, file, or inode from which to begin an investigation of an initial exploit, which is frequently not true. In the *prydain* example, there was evidence of changed files, but no conclusive evidence that these were part of the initial exploit.

4.2.3 Principle 3: Consider the Effects, Not Just the Actions

Failing to consider context can lead an analyst to draw incorrect conclusions from interpreting inputs. Interpretation of the input can be incongruent with the interpretation of the result, due to the way that changes in the user environment can affect translation of inputs. For example, considering only the pre-intrusion vulnerabilities and the post-intrusion system state cannot reliably give conclusive evidence about the astronomical number of ways in which the transition between the two may have occurred. Of the available tools, only ReVirt follows this principle, because it obviates the need to determine any relative importance of system events and objects ahead of time. By re-creating everything, the analyst can decide after the fact what is important enough to investigate and analyze in more detail.

Standard UNIX process accounting does not show how the context (the argument) affects the behavior of the program, and it tells an analyst nothing about the results of running that program (violating principle 3). *Keystroke logging* through the UNIX kernel's `sys_read` syscall shows user inputs, but not necessarily the *results* of those keystrokes (also violating principle 3). These limitations demonstrate that in forensics, *the result is at least as important as the action*, because if the user environment had been modified to change the effect of a keystroke, the action or input may be different than the result of that input. There are many other variations on this kind of behavior. In the *prydain* example, if the intrusion resulted from a remote exploit, these forensic mechanisms would have revealed nothing about the exploit, because no new processes were started, nor were any keystrokes input. Similarly, BackTracker does not consider context and environments along with the events that it records. Therefore, there is no easy way to determine the effect on the system of files opened, processes started, or network sockets connected.

Though BackTracker can show a series of kernel-level objects and events leading to the creation of a particular file, it is merely showing a part of what happened, and not what *could* have happened at a particular stage, nor what vulnerabilities existed there. In the *prydain* example, BackTracker may have shown that a shell was obtained following a Linux-PAM exploit, but what other actions could have been taken? Though BackTracker may help a forensic scientist decide where to look for a particular action or exploit, it does not necessarily help the analyst understand the nature of the action itself. For instance, process accounting may show evidence of a startup script running, and BackTracker may show a chain like `socket->httpd->sh`. This indicates that a shell was obtained from `httpd`, but does not help an analyst understand the nature of the vulnerability that allowed the shell to start. One way to analyze this is to ask the question at each stage: "What *could* happen here?" This approach allows an analyst to consider not only what is shown but also what else could have been done. Much like playing chess against a computer system and being able to see the computer's legal moves, this presentation should include a list of the intruder's actions, the *results* of those actions, and a tree showing other possible actions at each stage. Such a technique assists not only with forensic analysis, but also *vulnerability analysis.* We want to know not only the events that transpired, but also the context and conditions set up by those events, and the capabilities achieved by the user in each condition [TL00]. In the *prydain* example, we want to know not only the exploited program, but also the nature of that exploit and what else might have been vulnerable.

4.2.4 Principle 4: Consider Context to Assist in Understanding

Knowing the context helps an analyst understand when the result of an event may be incongruent with the expected result. Context also gives meaning to otherwise obscure actions. For example, knowing the user ownership of memory regions helps us understand if a user is attempting to share data with another user through a memory write. Knowing that a file is already opened by one program when another program attempts to write to it, tells us about a possible attempt at a race-condition exploit. We can record a user typing `setenv` or `chmod`, but current tools do not record the *state* of the targets of those commands, such as execution paths or file permissions, which could help an analyst understand the result.

We define an *abstraction shortcut* to be an event that bypasses or subverts a layer of abstraction on a system to perform actions at a lower level. Doing this makes events harder to understand. It is a common tactic, and many existing tools are confused by such tactics because they do not capture the different layers of context. For instance, although BackTracker does a good job of avoiding the potential problems arising as a result of tracking only filenames and not inodes, or processes and not process IDs, it does not compare UNIX group names and group IDs, UNIX usernames and user IDs, and data read or written to disk by reading or writing directly to the raw devices in `/dev`. This bypasses the UNIX file system. Finally, even where existing tools tell us that some data has been written to a raw socket or to the virtual memory residing in swap space on disk, they do not tell us what that data was, where in higher layers of abstraction it was written to, or what the *results* of the actions were. Context is a key element here. When an address is given, is it physical or virtual, and is that space in memory or in swap space on disk? The ability to translate raw disk read and write actions to something human-readable, such as its equivalent file on the filesystem, is essential. Neither BackTracker nor BSM consider context. The principle of using context to help understand meaning says that we can derive meaning from data if we know how it is being used and manipulated. For example, in the *prydain* example, context in the form of the addresses of the memory allocations and writes may have helped understand the exploit.

4.2.5 Principle 5: Present and Process Actions and Results in an Understandable Way

In general, forensic tools are not designed to do analysis. None of the standard tools installed on the machine in the *prydain* example do any analysis. Even the "enhanced" tools, which require being installed ahead of time and are supposed to aid a human to do analysis, do so either poorly or in passing, and the one model-based approach [Kup04] mentioned previously does not address analysis at all. Most tools collect or display information so humans can attempt to perform analysis. But the tools do little to analyze it themselves. The best tools only display collected information, and do not analyze, limiting their ability to present information in a coherent way. For instance, we can record keystrokes but we do not necessarily know the result of their entry [Tho84]. We can record and view a chain of processes, but we do not know what took place within them at the level of memory accesses, and we do not know the results of other operations that depend on the current state of the user space environment. We need to analyze and correlate recorded events using enhanced logging techniques, about both kernel and user space events and environments, enabling an analyst to distinguish meaningful results from the actions that caused them and the conditions which permit them.

Of the existing tools, only BackTracker attempts to post-process information to display it in a way that is more human-understandable. However, BackTracker requires a specific UNIX process or inode number from which to "backtrack" to an attack entry point. This limitation is severe when forensically analyzing an insider case in which there is no suspicious evidence or non-authorized activity to start with. Instead, it would be desirable instead to be able to analyze a range of times. In the *prydain* example, an intruder may have already been in the system for a considerable period of time before being detected. So if one particular suspicious object is found, the intruder may have already installed and used multiple backdoors after the initial exploit. BackTracker does not make it easy to discover these.

Only ReVirt follows the first two principles by enabling the re-creation of all events. However, the non-deterministic events that ReVirt records are analyzable only with great difficulty. ReVirt enables more analyzable information to be logged after intrusion during replay. But even ReVirt is not a complete solution to forensically analyze a system because it does not address principle 5. Better information and automated presentation and analysis tools must be gathered and developed.

In IntroVirt, each scenario (such as syscall race conditions) is approached as a service that must be implemented and run separately. But our forensic principles say that all the information should be already processed, synthesized, and readily available for query and analysis for any scenario. However, even if IntroVirt recorded every assembly code "store" instruction, the generated data would be worthless without careful post-processing that considered context (principles 3 and 4), such as the size of the memory allocation at the location being written to. From this point of view, IntroVirt does not have the proper goals for complete forensic analysis, as it does not follow principle 5. As IntroVirt runs below the guest operating system, in the *prydain* example, IntroVirt would have observed the entire intrusion, from original entry to subsequent entries and activities. However, because it does not monitor memory events, or support detailed correlation, translation, and automated analysis, it would not have been as effective as we desire.

4.2.6 Summary of Current Problems with Forensics

In the *prydain* example, the standard tools did not present evidence of the intrusion in a coherent way. The evidence was scattered throughout syslogs and the filesystem in an unorganized and uncorrelated manner. The standard tools for forensics simply fail to help analyze intrusions sufficiently in the general case, as well. Following the five principles that we have outlined is a possible solution to the current shortcomings. Currently, the standard tools do not address the entire operating system; they assume that user actions will generate system log events, keystroke evidence, or kernel event evidence; and they all completely fail to understand results as opposed to merely actions. If the standard forensic toolsets did address these areas, we have described how they would have vastly improved our understanding of the *prydain* example. In the next section, we suggest possible solutions that follow from the principles.

4.3 Principles-Driven Solutions

The primary gaps identified in the previous section are the lack of current tools that consider and synthesize data from user space, context, and results, as well as the lack of automated analysis. In this section, we present techniques that can be used to implement improved solutions.

4.3.1 Principles-Driven Logging

This section discusses a set of proposed techniques that follow from the forensic principles. Thus our goal is to propose techniques that might address current failings in forensic capabilities.

We begin outlining principles-driven tools by using selected techniques from existing tools. Principles-driven tools must record the nature and timing of interrupts and traps to the kernel (including syscalls and their parameters) as well as output from the kernel and information about asynchronous syscalls that have been interrupted (by an interrupt) and restarted. Tools must record memory allocations in both the stack and the heap, including their origins and timings. They must also record events involving other standard interfaces, such as the filesystem and network stack, including opening and closing of file handles, disk reads and writes, packets sent, DNS queries, and their precise timings. But this covers only a subset of common events needed to perform a forensic analysis. Therefore, we must address *all* of the forensic principles to maximize the possibility of producing better tools that possess methods of capturing and analyzing the remaining, necessary forensic information.

To satisfy principle 1, tools must record events in user space. These events include memory reads and writes (and their origins, contents, sizes, and timings), and also the names and types of function calls and parameters. The former would have helped to confirm the suspected remote buffer overflow exploit in the *prydain* example. The latter would have told us significantly more than existing tools. A principles-driven tool may be able to predict behavior based upon analysis of the program and certain memory events. For example, certain memory events performed inside frequently called, tight program loops may not need to be recorded, or recorded completely, since the same data should be gathered with lower overhead using other methods.

To satisfy principles 3 and 4, tools must also record *context* of both the kernel and user space, building a finite state machine in the process. In user space, context information is program-specific, including the shell, common applications, memory, and general user environment. While it is impractical to instrument every program on the system, some programs can be instrumented to save only memory events, and a very small subset of commonly used programs can be modified to save additional information. This can significantly clarify the results of actions by common programs. In the *prydain* example, the intruder may have made extensive use of the login shells, editors, or other common programs. The following is an example of the contextual information that could be captured to satisfy principle 3 and 4:

- operating system: users, groups, ownership, permissions
- login shell-specific: application execution paths, library paths, user limits, current working directory, keystroke mappings, and command aliases
- all programs: names of functions called, parameters, and names of variables read from and written to
- specific programs: application environmental information, including working directory, command macros, and other actions (e.g. from *vim*, *emacs*, or the *X Windows* environment)

From an implementation perspective, several methods can capture information about events that occur in user space, and choice of one is primarily important insofar as it satisfies auditing demands. One is *introspection* or monitoring of a virtual machine. This technique has been used successfully [GR03, JKDC05] with security and allows the host operating system to monitor everything that occurs in the virtual operating system. A key benefit over other solutions may be a relatively low performance overhead. Unfortunately, introspection of a running virtual machine for the types of events that are of forensic interest is likely to increase performance overhead significantly. This is because capturing high-frequency, low-level events, such as memory accesses, would then suffer the penalties of high-frequency introspection in addition to any performance penalties from using a virtual machine. Therefore, the greater the performance penalties from recording the necessary forensic information, the less useful it is to use a virtual machine, which would add its own performance penalties. Likewise, as mentioned earlier, introspection upon replay suffers from the problem of interfering with the events being replayed, or replaying imprecisely if hardware conditions changed.

Other solutions for capturing user space information do not use a virtual machine. Programs are built by compilers that can capture additional information, both at compile-time and run-time, about the programs. Binary rewriters can instrument binaries to record and save run-time logging data. Programs run with an instrumented compiler or binary rewriter-like tool can tell us, for instance, the nature of dynamic code written by a user program onto the heap and executed at runtime, as the instrumented programs can record what is written to memory and executed.

NetBSD 2.0 contains a feature called *verified exec* [Lym04] that can be used to impose restrictions on running only cryptographically-signed ("fingerprinted") binaries. In this way, having forensically valuable information compiled into binaries could be enforced. In the compiler-instrumentation approach, user space information can be recorded by instrumenting the system's C/C++ compiler and mandating that any binary run on the system, including the kernel, be compiled with the special compiler. This approach also has the benefit of saving more user-understandable information than virtual machines or binary rewriters because it can force all binaries to have debugging and profiling information compiled in. With binary rewriting, a binary can be instrumented to gather information similar to that which a compiler can give. The implementation is simpler, but the presence of the symbol table cannot be guaranteed.

One drawback to these approaches is the amount of information that would be generated. The approach of instrumenting a compiler, as opposed to using a binary rewriter, could significantly reduce the amount of data necessary. For instance, to investigate buffer overflows, new tools need to capture all `sbrk` and `mmap` syscalls, as well as capture sizes and timing of memory writes to those allocated variables. However, it is likely that new tools will *not* need to record *all* memory writes. Assembly code *store* instructions generated by the compiler for manipulating intermediate variables could represent a massive portion of the code, and these do not need to be recorded. Unfortunately, a binary rewriter does not know how to deal with any higher-level constructs. On the other hand, after recording the syscalls above, a compiler could insert code not after every assembly store instruction (unless there is assembly inline with the C/C++ code) but after every C/C++ assignment operation, as represented in the compiler's abstract symbol tree (AST) or other intermediate language. While there are also a very large number of assignment operations in typical C/C++ code, the number may be an order of magnitude less than the number of assembly code store instructions. Therefore, though using a binary rewriter is undoubtedly less cumbersome than instrumenting a compiler, the improvement of the resulting information given by both the symbol table and the ability to audit events and constructs at a higher level than assembly code, is likely to improve forensic analyzability. The timing, size, and nature of memory writes is merely one example of this.

Using compilers and binary rewriters raises the following problem: capturing information about the entire system requires that even the operating system be recompiled. If imposed on the kernel and drivers, this restriction could cause problems for code dependent on specific timing responses from the hardware. While not all systems have such dependencies, we would like our techniques to be general. Fortunately, the parts of the code that rely on timing information interact with the hardware and need not be instrumented at the same level as all the other system and user code. Because we know the inputs to the kernel (syscalls, traps, interrupts), and the kernel itself, are deterministic, we can determine the result of the inputs to the kernel via replay. Similarly, by using forensic data gathered from other programs and by drawing upon collected user space information (for instance, a hash of the memory image of the kernel [GR03]), we can determine how a kernel has changed and how that change has affected the system, without instrumenting the kernel itself.

To date, we have made a conscious decision to concentrate on completeness and efficacy rather than efficiency and performance. Obvious approaches to improve performance include information compression, co-processor-assisted logging [PFMA05], and dedicated hardware [XBH03, NPC05, CFG+06] for logging non-deterministic events.

In the next section, we discuss principles-driven auditing, particularly principle 5, and specifically how to audit the information obtained using the techniques described in this section and present it to an analyst.

4.3.2 Principles-Driven Auditing

Though only represented by a single principle, the most difficult part of forensic analysis is auditing the data. A solution requires a method of presenting kernel and user space context and events together to the analyst. New tools do not need to exhaustively *analyze* the data themselves. This is because doing so would require foreknowledge about the nature of the event, which is an assertion that we avoid (principle 2). Rather, a principles-driven tool should exhaustively *log* relevant data, and, in presenting it, enable the human analyst to perform analysis more easily and completely. A presentation should include the raw events themselves, and enable the ability to easily view events and environments at arbitrary points in time. It should also allow correlation of those events arbitrarily with others at similar points in time or operating on similar points in memory or on disk. An analyst should be able to easily speculate about global questions involving forensic data ("were there any potential memory race conditions recorded in this day-long time period, and where?") and also to look in more detail [Gro97] into the macro-views of process and file information provided by existing tools to find answers.

Storing and representing data in a coherent way is critical to the forensic process. A first-order step for auditing is correlating and associating all recorded objects and events into a multi-resolution, finite state machine. To analyze this information, new tools can use techniques similar to those used in debugging, which allow a programmer to "step" into functions or walk through higher-level function calls. We define this display of detailed information along with coarser-grained data to be a *multi-resolution* view of forensic data. This would allow an analyst to zoom in on specific processes and memory events, and anything that those events are related to, to see more detail. In keeping with this goal, tools must *store* data in a way that enables this. One way is by viewing a computer system as a relational database. An application launch can be viewed as a record in a table, having a large number of items associated with it. At the least, this includes: a process ID, user ID, group ID, time, checksum, path, current working directory, size of initial stack memory allocation, set of heap allocations, set of functions, set of variables, and a set of filehandles associated with the process. Each field within this process record is also a record itself. For example, a user ID needs to be associated with a user name, and also with processes, file handles, heap allocations, memory writes, and so on. Using symbol table information, each memory allocation is associated with a variable, function, program, user, and time. A critical part of the automated analysis is translating abstract addresses into understandable objects and events (principle 4).

Principles-driven tools can perform context-assisted translation not only for memory but also for abstraction shortcuts. For example, a write to an arbitrary disk location through a raw device may have a file associated with it (and if it does not, this can be an indication of covert information sharing). Then, the same correlations that were done with memory can also be done with files and network events: a file or socket is owned by a user and has a process ID of the process which accessed it, and so on. All of this correlated information, including how data is viewed or modified, should be in the multi-resolution view.

Once translations and correlations are finished, principles-driven tools can perform automated analyses to generate warnings for a human analyst. As with intrusion detection [Den87], automated methods can be both anomaly-based and signature-based. Anomaly detection is always an available tool, but is useless when the "attack" in question is "normal," common, or innocuous enough not to appear as an invariant. However, if modeled correctly, even those can be discovered through signature detection. Usefulness of anomaly detection and signature detection to forensics may be exactly the inverse as their uses to intrusion detection, because in forensics, we do not have to predict signatures in advance and can refine them after the fact.

An example of the result of an anomaly detection-based approach using statistical or artificial intelligence techniques might involve discovery of an invariant that indicates that a user's files were modified by the superuser. If that action is rare, then an anomaly is indicated. An example of a signature detection-based approach is a tool that can compare the sizes of memory writes to buffer sizes to look for potential overflows. Rapid accesses on the same location in memory or disk by different programs should provide a warning about a possible race condition, as would rapid accesses to the same network ports. Additionally, addresses of memory writes by user programs can be audited to see whether the program instruction stream is being tampered with, and correlated with user IDs to determine whether information is being shared between user spaces. Those same memory writes can also be analyzed to determine whether they might be covertly recording user inputs. Program environments at the time of each event can be audited so that effects of actions can be correctly identified. Function names can be analyzed in an attempt to determine whether remote access may have been granted by programmer backdoor or by exploit of a software bug. Tools can also provide a facility to keep a record of the history of the values of selected variables in memory, and when different programs accessed or changed them.

Covert channels are difficult to eliminate, and even where they are preventable, often it is undesirable to do so. On the other hand, *overt storage channels* and even *legitimate channels*, are currently as badly audited in today's operating systems as covert channels. A principles-driven forensic system would possess the necessary data, since it logs all such data.

A final method of analysis required for principles-driven tools, addressing principle 2, is to perform analysis not only about what *did* happen, but what *could* have happened at each step in a system's execution, both in kernel and user space. For example, in a buffer overflow, the return address is typically altered to return the execution point to an alternate location. In this automated analysis, we also want to know the other active programs and their functions that could have been jumped to. Or, in a possible race condition situation, we want to know the programs and nature of the objects involved. To perform this analysis, one might use requires/provides techniques [TL00] as a model to present abstract events, and look not just at that series of events but also at a set of *conditions* and *capabilities* acquired given the events and the context in which they occur.

The approach we have described in this section addresses one possible solution to principle 5. They completely transform typical methods forensic software uses to present information to a user. No current tools come anywhere close to providing *any* sort of useful and automated analysis without sacrificing a significant amount of accuracy by ignoring or filtering out relevant data. Combined with proper data acquired by adhering to the first four principles, these techniques give a possible solution to performing forensic analysis in a way that assists a human analyst to obtain proof, not inference, in a practical way.

4.3.3 Summary of Principles-Driven Solutions

Using either introspection of a hypervisor, a binary rewriter, or compiler modifications, principles-driven tools must gather not only kernel events, but also information on timings, sizes, and locations of memory allocations, reads, and writes. Tools must gather information on events using abstraction shortcuts, particularly those bypassing the filesystem or network. They must gather information on program, function, and variable names. By correlating those names, memory events, system context, and program environments, principles-driven tools must translate these objects and events into human-understandable data. Finally, after generating human-understandable data, principles-driven tools must present that data in a *multi-resolution* fashion that allows for viewing data at granularities ranging from memory writes to program launches and user logins. This representation should provide an opportunity for automated vulnerability analysis of not only what *did* occur but what *could* have occurred.

4.4 From Principles to Models

The principles of computer forensics help us devise techniques to significantly improve our ability to understand what has happened previously on a computer system, when compared with current tools. Those techniques do not require pre-determination of the nature of the events or the skill level of the attacker, and do not require the analysis to begin with knowledge of precise details after the fact about users, times, processes, and system objects involved.

We believe that looking at the complete system to record information not recorded by previous forensic tools (principle 1), particularly data about user space events and environments (principles 3 and 4), and events that have occurred using *abstraction shortcuts* (principle 3), will allow us to more precisely analyze events that involve covert memory reads, buffer overflows resulting from memory writes, race conditions in memory or on disk, reads and writes to raw devices, and other similar events. These techniques address forensics without making assumptions about the opponent (principle 2), and they allow for understanding not just actions, but the results of those actions based on context (principles 3 and 4). Auditing tools that allow for analysis of the recorded information should also allow for vulnerability analysis based on the current context from any point in time, translation of abstraction shortcuts to a higher granularity, and, most importantly, a multi-resolution view of the data, which allows zooming in and out of kernel and user events and environments, and the ability to easily analyze at any point in time (principle 5).

We have also shown how techniques derived from the principles have the potential to perform their work in a practical way, as demonstrated in Chapter 3 using the function call tracing approach. We can see how adhering to principle 1 and collecting function calls at runtime and within within user space aided in the analysis of *prydain* example. Without that information, analysis would have been significantly more difficult, than, for example, looking at system calls alone. Similarly our technique would not have been as successful if we looked only at single calls (that is, $k = 1$). The fact that this approach looks at *sequences* of function calls, and therefore largely addresses both cause and effect, as desired by principles 3 and 4, significantly aids in its effectiveness. Finally, one of the key benefits of applying intrusion detection techniques to forensics is to aid an analyst in understanding the information by highlighting the suspicious parts of the execution. This feature, inspired by principle 5, significantly aids a forensic analyst by reducing the amount of data that the analyst needs to look at in order to analyze an attack.

The techniques derived from these five forensic principles also lead to answers more easily proven correct. This greatly reduces inferences and guesswork. These concrete answers are exactly what we desired in the *prydain* example, described in Chapter 3, and were impossible without the changes that that we suggest. The techniques contribute towards making the final analysis by a human easier by performing automated analysis first. And finally, they contribute to addressing scenarios that were previously unsolvable, such as the insider problem and events occurring in user space.

Our method using sequences of function calls demonstrates that the principles can, for the most part, be applied in a practical way. However, the solution using sequences of function calls is still *ad hoc*. Thus far, the solution has largely been fine-tuned to a small set of attacks, and the effectiveness of the solution and the data needed (such as function returns or call parameters) varies unpredictably depending on the nature of the attack to which it is applied.

The absence of a rigorous approach to forensics indicates the need for a *model* from which to extract the exact logging requirements. To the best of our knowledge, such a model does not currently exist. We use these principles, as well as several qualities that we believe a good forensic model should possess, to guide the construction of a model.

4.5 Qualities for a Forensic Model

In addition to adhering to the forensic principles, a good forensic model should have several other qualities:

As recording every event on a system is impractical, a forensic model should indicate the information necessary to log, but let system administrators choose whether to record the information or not. One way to do this is to focus on particular paths in an attack graph. A number of metrics could aid in this simplification. For example, potential paths could be ordered, and paths with fewer than n possible exploits or paths longer than s steps could be placed low in the ordering. The metric we currently use is *severity*, which we derive from the credentials obtained, and which range from no local login at all to superuser-level access.

An attempt to alter a file can cause log entries at the application layer, the library function layer, and the system call layer, and each particular action would have a unique tag that propagated from the highest layer of abstraction down to the lowest. That way, the analyst could instantly determine the exact system calls used to (for example) write data to a file, and verify the data passed to the library function was in fact passed to the system call without alteration. Therefore, a forensic model should indicate places that might require modifying a system to log at intermediate layers of abstraction, or multiple levels of abstraction.

Making assumptions about expected attacks or the abilities of an expected attacker can limit the abilities of a forensic system. But the process of modeling *all* possible attacks on *all* possible systems that enable an attacker to gain certain capabilities on those systems is intractable. Therefore, modeling unknown intermediate goals by placing bounds on the path between known goals can give information about the attack that occurred in between. So, a good forensic model should describe, or put bounds on, portions of an attack graph that have many possible paths, to show what needs to be recorded to disambiguate the paths.

The bounds on a goal also help an analyst understand the effects of events as well as the actions that caused them. Hence, a good forensic model should consider both the conditions of the system before and after a goal in an attack graph is accomplished. By identifying the the most important contextual elements surrounding an event and recording them, the method that an attacker uses to achieve a particular goal might also be more easily understandable.

Finally, a forensic model must enable analysis of multi-stage attacks by requiring that information is well-formed enough to be correlated and associated with other discrete events that comprise a larger attack [ZHR$^+$07]. Further, and perhaps most importantly to a rigorous model, a forensic model should be composed of translation functions that are as close as possible to *one-to-one* functions, so that sequences of logged events can easily be inverted to describe unique sets of events, or at least unique classes of events.

To summarize, a good forensic model should possess the following qualities:

1. The ability to log anything.

2. A provision for automated metrics, such as path length, and a tuning parameter that enables system administrators and forensic analysts to decide what to record.

3. The ability to log data at multiple levels of abstraction, including those not explicitly part of the system being instrumented.

4. The ability to place bounds on, and gather data about, portions of previously unknown attacks and attack methods.
5. The ability to record information about the conditions both before (cause) and after (effect) an event has taken place.
6. The ability to model multi-stage attacks.
7. The ability to translate between logged data and the actual event in a one-to-one fashion.

In the next chapter, we describe our specific model based on these qualities, and a methodology of applying and using it.

The work presented in this chapter was first described in an two earlier papers. The first paper was, "Principles-Driven Forensic Analysis," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, in *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, pp. 85–93, Lake Arrowhead, CA, October 2005. The second paper was, "Toward Models for Forensic Analysis," Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, to appear in *Proceedings of the 2nd International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*, Seattle, WA, April 2007.

5

Laocoön: The Forensic Model

Sherlock Holmes: *"The world is full of obvious things which nobody by any chance ever observes."*

—Sir Arthur Conan Doyle, "The Hound of the Baskervilles," *The Strand Magazine* (1902)

5.1 Introduction to Our Approach

This section introduces a forensic model called *Laocoön*,[1] that possesses the qualities that we have established in the previous chapter as being helpful in forensic models. Our model builds upon recent work in formalizing multi-stage attacks — for the purposes of intrusion detection [TL00, ZHR$^+$07] — to develop formalisms that allow us to derive rigorously what is needed to log for forensic analysis.

The *requires/provides model* [TL00] considers an attack as consisting of a series of *goals*. Each *step* in the attack achieves an *intermediate goal* that advances the attacker closer to the *ultimate goal*. Further, any of a number of actions may constitute a step. For example, if an intermediate step is to read a file, an attacker may use any of a system pagination program, a text editor, or a specially crafted program to read the file. The exact technique used is *not* explicitly modeled. Achieving the intermediate goal *is* modeled. This model represents attacks as graphs, the intermediate goals being intermediate nodes, the ultimate goal being the root of the graph, the leaves being starting points, and the edges being steps in the attack.[2] By *inverting*

[1] *Laocoön* was the unfortunate Trojan who recommended not letting the Trojan horse into Troy (and was strangled by divine serpents for his investigative and civic efforts).

[2] This is essentially a formalization of attack trees [Sch99], commonly used for intrusion detection.

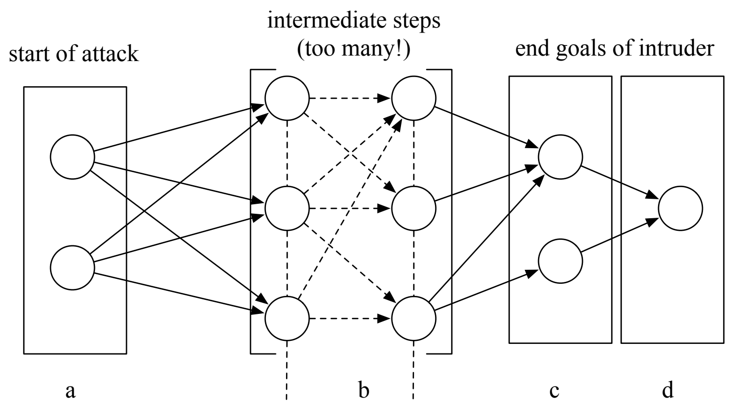


Figure 5.1: Diagram of a generic attack where circles represent actions. An attack model almost always consists of at least the endpoint (d), but may also include the beginnings (a) and possibly other states near the end (c).

the requires/provides analysis, one can move from a goal to a set of paths through the attack graph, each path representing a possible attack to achieve that goal.

Figure 5.1 shows an attack graph. We work backwards from the single, ultimate goal of the graph due to convergence of methods, either at the start (Figure 5.1a) or end of an attack (Figure 5.1c/d), as opposed to the explosion of possible methods that could be used in the middle of an attack (Figure 5.1b). Indeed, an intermediate goal could even be part of several attack graphs.

The methodology for using the requires/provides model is as follows:

1. Choose a set of attacker goals (starting, intermediate, and ultimate) to model.

2. Working backwards from the ultimate goal, build requires/provides models of one or more goals in each exploit.

3. Specify the pre- and post-conditions ("capability sets") of each goal, outlining the set of data to be logged. We represent each condition using a 6-tuple, defined below.

4. Because some intermediate goals are not known, use the pre- and post-conditions of the known goals to determine what the intermediate, unknown goals need to be, and for each, derive the appropriate 6-tuples, as in bullet #3.

5. Finally, from the pair of capability sets ("capability pair"), extract the information that needs to be logged.

We now describe each of these steps in detail.

5.2 Choosing Intruder Goals to Model

Goals are different than vulnerabilities, attacks, or exploits. While attacks are unpredictable, many intruder goals are already known. Further, vulnerabilities can appear and disappear, based on software bugs or on temporary configuration errors, but goals remain [BCF+97].

There have been many research efforts to address multi-step attack modeling. Some have used ad hoc approaches to generate graphs, such as probability, or pre-conditions and post-conditions designed by "red teams." [VVKK04]. Other graphs have been based on specific exploits, or known vulnerabilities [CLF03, CM02, MMDD02, NCRX04]. Two of those in particular have used rigorous model checking approaches [OBM06, SHJ+02] to translate vulnerabilities (or system configurations that allow vulnerabilities) into attack graphs. But these methods are insufficient for our needs because they limit the amount of information collected to that which is known and can be predicted exactly in advance.

Our objective is different than these existing approaches because our model is directed at forensic analysis, not intrusion detection. As a result, we collect information necessary to understand and analyze an attack, not the information that would simply indicate the possible existence of an attack. In many cases, simply indicating the existence of an attack leaves an analyst helpless to actually understand the attack. Forensic analysis answers *what* and *how*; intrusion detection answers *if*. The difference is important. For forensic analysis, we need not only the information indicating an attack has taken place, but also the information that describes *how* it took place and *what* damage was done afterwards. Many possible scenarios are *not* recognized by any intrusion detection system as an attack, because the series of events might only be an attack some of the time, but not always. Our research efforts attempt to determine what information is *necessary* to analyze an attack. It does not assume that the information recorded to answer the *if* question is sufficient to perform a thorough analysis. So, we must construct our goals based on specific conditions that help us log appropriate data. This approach suggests the controls that need to be implemented to reconstruct the attacks. This is done by analyzing the goals and determining the steps an intruder took to achieve those goals.

High assurance systems develop logging and auditing subsystems based on formal methods.[3] The formal methods instantiate precisely defined confidentiality and integrity policies such as Bell-LaPadula [BL75], Clark-Wilson [CW87], and the Chinese Wall [BN89]. The approach of developing forensic subsystems based on the formal methods uses the axioms put forth by the policies to dictate logging requirements. However, real-world policies mix implicit elements with explicit elements, and are inherently ambiguous.[4] A solution to this might be to implement

[3]See §24.3, "Designing an Audit System," in *Computer Security: Art and Science* [Bis03].

[4]However, a very limited number of real-world, practical systems *could* make use of this, such

models based on *discovered security policies* [BP06] or well-defined vulnerability classifications [Bis99], however, goals could also be more arbitrarily defined.

Our solution expands the approach built on modeling classical policies to real systems, while also using a more flexible and better-defined basis for the logging. In this chapter, we describe the data necessary to analyze certain goals. Ultimately the goals should not be constructed manually, but could use some of these existing techniques to be defined in an automated fashion. However, we do not yet know what data is necessary to do this, and since our process is not automated, we cannot yet directly compare the effectiveness of our approach to existing methods, such as our method of performing forensic analysis using sequences of function calls, described in Chapter 3.

5.3 Modeling Intruder Goals

The requires/provides model describes the rights and information required to achieve a goal, and the rights and information that the successful attack provides. For example, in Figure 5.1, the "requires" conditions represent the capabilities immediately before the circles (actions) and the "provides" represent those immediately after. The figure shows a series of steps towards an ultimate goal to be analyzed.

The goal of the analysis is to determine the appropriate attack graph and the paths in the graph that represent the steps that the attacker used. Approaching the problem of forensic reconstruction of events through this light requires developing a set of possible ultimate goals for the attacks, for example by traditional techniques of threat analysis. Then, the possible ultimate goals suggest attack graphs culminating in those goals. From these graphs, the analyst can ask what changes in the state of the system each attack step introduces. The answer provides the information needed to detect the change (or to determine that the change did not occur). If the information is available, the analyst can obtain it. If not, the system can be augmented to record it. This ties forensics to a model of attack, and provides a basis for collecting specific information.

The forensic analyst is often unsure of the ultimate goal of the attacker. Indeed, when we refer to an "ultimate goal," it may merely be a single "ultimate goal" of many desired goals of an attacker. Figure 5.1 shows a single series of steps towards a goal that we wish to log data about, but a series of actions may actually look like Figure 5.2. An analyst typically determines that the system has been attacked because she finds information indicating an intermediate goal

as ones based on the *Take-Grant Protection Model*, because with the Take-Grant model, the security question is decidable [JLS76], whereas in most systems, it is not.

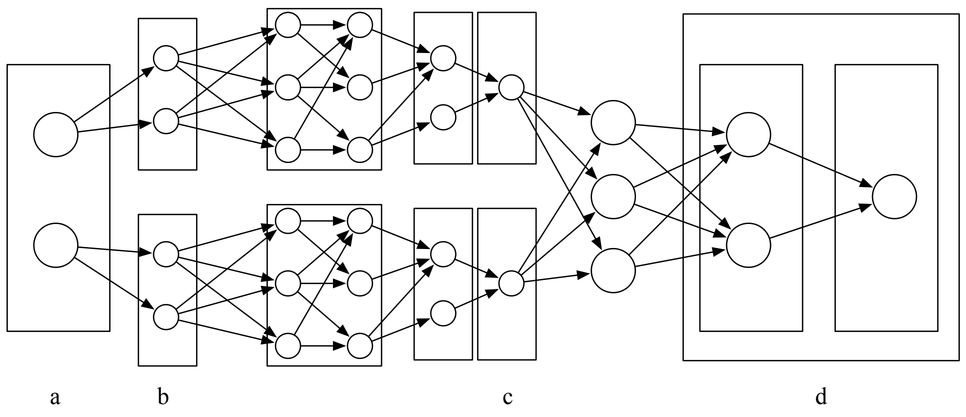


Figure 5.2: A coordinated, multi-stage attack. (a) represents a dual-pronged, coordinated beginning of the attack, (b) represents the beginnings of two individual components of the attacks, (c) represents the "ultimate goals" of each individual attack, and (d) represents the ultimate goal of the entire attack.

has been attained. The intermediate goal could be part of several attack graphs. The goal of the analysis is to determine the appropriate attack graph or graphs (that is, the ultimate goal) and to determine which element of the set of paths within the graph (or set of graphs) represents the attack steps that the attacker used.

Capturing precise exploits used to achieve intermediate results may rely upon recording events that are too low-level to capture or require saving too much state information (e.g., memory reads and writes [PBKM05]). However, by piecing together the results of attempts at achieving the end result and preceding intermediate goals, we may be able to infer the exploit with a high degree of confidence.

There are multiple ways to describe a requires/provides model. One way is using the JIGSAW language described in the original requires/provides paper. Another way is by using the method developed by Zhou, et al. [ZHR+07], as we repeat here:

The requires/provides model contains a notion of "capabilities" to describe a set of attributes that are *required* to attain a goal, and another set of attributes that reaching a goal *provides*. That is, the "capabilities" are predicates that represent the pre-conditions and post-conditions of the goal. The analyst can look for indications of those capabilities. If those indications exist, then an attacker attained the goal to which they apply. At this point, an intermediate goal has been found, and the analysis proceeds as described above.

Note that "capabilities" are not the same as "implications." A *capability* is something that is immediately gained through a step in a process. An *implication* represents theoretical possibilities that could be done given certain information. For example, a "capability" might be knowing a password. An implication might be the ability to log in to a system using the password. Our model addresses the former, not the latter.

Let L be a set of locations, A a set of actions, C a set of credentials, S a set of services, P a set of properties, and $R = L \times L \times A \times C \times S \times P$ a set of capabilities. A *capability* is a predicate asserting that the *source* takes an *action* based on the *property* of the *service* using the *credential* on the *destination*. We represent this as a 6-tuple:

$$(source, destination, credential, action, service, property) \in R$$

Previous work in intrusion detection [ZHR+07] uses this structure for capabilities because the information described in those fields is obtainable from common audit logs. Fortunately, these fields also serve our needs. For other needs, capabilities could be defined as appropriate. For example, if distributed shared memory were the conduit, the address would actually refer to the shared memory addresses.

Each *goal* is modeled by a *capability pair*, which is an ordered pair of *capability sets*. One set describes the conditions required to accomplish the goal, and the other describes the conditions provided by reaching that goal. There can be more than one capability in a requires or provides set to describe multiple conditions that must be met before a goal is achieved, or multiple capabilities that are provided by achieving a goal. Where E is a goal, the first element in the pair is a set of "requires" capabilities and is notated as $E.requires$. The second element in the pair is a set of "provides" capabilities, which is notated as $E.provides$.

In a capability, *source* and *destination* refer to addresses. In this dissertation, our examples all contain IP addresses. A *credential* is some level of access control. It could be a uid or gid, if local access is considered, or it could be an IP address in a specific subnet if remote access is considered. Or it could simply be a valid login or network connection. In most cases, it refers to a uid of *root*, a (specific) *user*, *no* user, or ANY user.

Table 5.1: Possible Actions

Action Type	Action Values
Read	read, list, know
Write	create, modify, append, delete, allocate, spoof, compile
Communicate	send, receive, connect, listen
Exec	invoke, exec, call, running
Block	block, delay, spoof

An *action* is an event such as the ones listed in Table 5.1.

Some of the the kinds of *events* (the real-world analogue to *transitions* that are possible to log on a UNIX-like system) include:

1. UNIX shell commands and program executions
2. system calls
3. dynamic library calls
4. function calls
5. network traffic
6. call (of any type) parameters and return values
7. assembly instructions
8. keystrokes

Table 5.2: Service Types

Service Type	Service Sub-Types
Hardware System	network interface, CPU
Software System	OS, DBMS
Process	daemons, user processes, OS processes
Account	ordinary users, root-like, root, single-user mode
OS Kernel	kernel modules, network stacks, system calls
File System	directories, files (programs, scripts)

Table 5.3: Service Property Types

Service Type	Property Types
CPU	architecture, processing capacity
Net Interface	MAC address, bandwidth
OS	version, patch level, running status
Network Stack	running status, address, sending, replying
System Calls	code
Processes	running status, version, port, protocol, option
Accounts	name, password, id, activity, shell, home directory
Directories	path, existing, permission, owner, timestamp
Files	link, path, existing, permission, owner, timestamp, content

A *service* is a level of functionality, such as a program, the filesystem, or a portion of the kernel (see Table 5.2). A *property* is the portion of an object (e.g. permissions, running status, version, path, or owner) that we might consider (see Table 5.3). For example, a user might *exec* (action) the *code* (property) of a *shell* (service), but might a user might also *modify* (action) the *permissions* (property) of that same *shell* (service).

The two special functions ANY and ALL denote one or all values in a set, respectively.

An analyst must also be able to analyze intermediate goals that he knows nothing about. Let A, B, and C be goals that must occur in immediately sequential order. Assume that A and C

are known intruder goals and B is an unknown intruder goal. We define a transformation function τ that takes a provides/requires pair as input, and outputs a requires/provides pair. That pair describes the beginning and ending capabilities of a set of one or more unknown, intermediate goals, B, that occurred immediately after A and immediately before C. So we define the function τ as a transposition of capability sets as follows:

$$\tau : (A.provides, C.requires) \rightarrow (B_{max}.requires, B_{min}.provides)$$

where

$$A.provides \equiv B_{max}.requires$$

and

$$C.requires \equiv B_{min}.provides$$

The resulting capability pair $(B_{max}.requires, B_{min}.provides)$ is then treated as an ordinary capability pair. We demonstrate the correctness of the τ function in Section 6.10.1.

$B_{max}.requires$ and $B_{min}.provides$ do not represent precisely what the goal B requires and provides. $A.provides$ must give the minimum number of capabilities actually required by B, but may in fact give more. Otherwise, a traversal of the attack graph could not proceed because achieving A would provide insufficient capabilities to accomplish B. Likewise, the capabilities required by $C.requires$ must be no more than those provided by accomplishing the goal B, but in fact, $B.provides$ may actually provide more capabilities than C requires. Otherwise, C could not be achieved. Hence $B_{max}.requires$ and $B_{min}.provides$ represent maximum and minimum bounds, respectively.

In a multi-stage attack, sub-goals and events can also combine to achieve a more complicated goal. For example, suppose that an attack involves two goals A, B that are both required to be achieved before a third goal D. We can model this in the same way that we model any other set of goals in an attack graph, except that D will have at least two 6-tuples in its requires set.

Suppose that there is an unknown sub-goal, C, that occurs after A and B, but before D. We want to be able to apply the τ function, but τ takes only a single provides set and a single requires set as input. In this case we have two provides sets and a requires set. We need a way of combining multiple provides sets, that is, in this case, $A.provides$ and $B.provides$, to obtain $C_{max}.requires$. To do this, we introduce the μ function. The μ function deals with correlated goals somewhat similarly to Ning's *correlated hyper-alerts* [NCRX04] and Zhou's *m-attacks* [ZHR+07]:

Let e be a set of two or more requires sets or two or more provides sets. Let e' be a single requires set or a single provides set. We then define the function μ as:

$$\mu : e \rightarrow e'$$

Let n be the number of sets in e. Then:

$$\mu(e) = \bigcup_{i=1}^{n} e_i$$

Returning to our example with the goal C after sub-goals A and B and before D, the λ function to obtain $C_{max}.requires$ and $C_{min}.provides$ then looks like:

$$\tau(\mu(A.provides, B.provides), D.requires)$$

$$= \tau(A.provides \cup B.provides, D.requires)$$

$$= (C_{max}.requires, C_{min}.provides)$$

```
BOUND-UNKOWNS ($\Sigma$)
INPUT: $\Sigma$ is a set of $n$ requires/provides capability pairs, $E_1 \ldots E_n$ describing an attack graph to achieve a specific ultimate goal, $E_n$ and, if $n > 1$, contains capability pairs for $n-1$ subgoals, $E_1 \ldots E_{n-1}$ The nodes in the $\Sigma$ are numbered from $n$ to 1 in a depth-first search (DFS) fashion, stopping and going back up to the top when $i-1$ has more than 1 connection (but not computing the $i-1$ node), unless all of the other nodes connecting to the $i-1$ node have already been looked at.
PRINTED OUTPUT: The attack graph, $\Sigma$, with the unknowns bounded
1 for $i = n$ to 1 do
2     if $(i \geq 3)$
3         if $(undefined(E_{i-1})$ and $defined(E_{i-2})$ and $numsets(E_{i-2} = 1))$
4             $E_{i-1} = \tau(E_{i-2}.provides, E_i.requires)$
5         else if $(undefined(E_{i-1})$ and $defined(E_{i-2})$ and $numsets(E_{i-2} > 1))$
6             $E_{i-1} = \tau(\mu(E_{i-2}.provides), E_i.requires)$
```

Figure 5.3: Algorithm for placing bounds on the unknown goals in an attack graph.

The algorithm BOUND-UNKNOWNS (Figure 5.3) is used to apply τ and μ to the goals in an attack graph that are unknown. The algorithm works by starting at the “ultimate goal”

— the last node in the attack graph — and walks backwards via a depth-first search. When the algorithms finds an unknown goal, it uses the known goals that precede and follow the unknown goal as inputs to the τ function, and outputs the capability pair for the unknown goal. In cases where multiple goals are required before proceeding to the next goal, the Bound-Unknowns algorithm takes the union of the capabilities required. Also, by definition, there cannot be more than one unknown goal in a row. Multiple unknowns are compressed into a single unknown.

5.4 Extracting and Interpreting Logged Data

One important question is how to log data derived from the capability pairs in a way that is practical and precise. For example, one intermediate goal may be to alter the password file. Existing systems lack the application logging mechanisms required. System level logging (most likely) represents this action as a sequence of system calls involving the execution of a program, the opening of the password database, the reading of a sequence of records, the writing of one or more records, and the closing of the file. These events are indicated partially through a large number of supporting system calls (such as dynamic loading and memory mapping for I/O). Thus, we need to take a high-level abstraction from our graph ("change a password") and derive the low-level implementation of that abstraction, so we know what to log.

Conversely, a reversal of this process enables us to aggregate low-level system call log entries into higher-level abstractions, thereby aiding the process of reconstructing intermediate goals. This has been done successfully using the requires/provides model for analysis of network intrusion detection logs [ZHR+07].

Sometimes log information can be either accidentally or intentionally distorted. Therefore, handling that bogus log information, or misleading data gleaned from the system, requires that the conclusions drawn from those sources be validated. Events cause changes to states or data; similarly, changes to data or state occur as a result of specific events. By cross-checking the data, context information, and state information with information about events, the analyst can determine to some degree the accuracy of both the events recorded in the logs, and the accuracy of the analyst's interpretation of the data and events.

Finally, pre-defining sets of both good and bad sequences of events enables the analyst to discard information quickly. For example, suppose a corpus of known, good sequences of function calls is pre-defined. An unexpected function call, or an expected function call with unexpected arguments, may indicate a step in an attack and enable us to eliminate a large number of potential paths. Therefore, by defining known, good sequences of calls, we can define a set of normal actions and activities that we can safely exempt from routine logging. Some such sequences may be information-independent — sequences of events that are always bad on a particular platform, regardless of a particular site's security policies [BP06]. Some sequences may be contained in an installation specific set of events constructed from recorded sequences of "safe" function calls over an extended period of time [PBKM07a].

The mapping between system-level logs and application-level logs is critical because of the end-to-end principle.[5] This principle states that logging should be done as close to the source as possible. In terms of systems, this means that logging should be performed either at the point of resource access (controlled by the reference validation mechanism or its equivalent, typically the kernel) or at the point of use of the resource (the application). Collecting the data at either point involves tradeoffs of performance and reliability against quantity. The set of information required to identify a path uniquely may be so great as to overwhelm storage or I/O processing, or the ability of the processor to perform other computations. Both of these extremes are unacceptable. Further, if arbitrary programs can log information at the application level, the logs could be populated by bogus messages;[6] in fact, syslog, a widely used logging mechanism on UNIX-style systems, is susceptible to exactly this attack. Thus metrics quantifying the benefits and drawbacks of recording everything, and modeling the integrity and reliability of the logs given particular configurations of the log files, will provide information to help administrators and forensic analysts determine the appropriate trade-offs.

The information to log falls out of the requires/provides capability pairs. We wish to generalize the formal process of extracting data from capability models. We apply an algorithm to each subgoal in the attack graph. Analyze-Goal takes the capability pairs of each goal as input. It outputs three items: (1) the information to be logged, (2), the mechanism on the system to intercept the information, and (3) where the information is to be obtained (the "logging point"). The source and destination addresses and credentials are used to select and constrain the data observed. The action, service, and property ultimately give much of the information about exactly how the goal was achieved. Ideally, this process is automated, but this is beyond the scope of this dissertation. As an example, the matrix below indicates a number of possible combinations of action, service, and property, and the resulting logging point for a UNIX-like system. However, in order to automate the process in the general sense, we would need a specialized policy language and the ability to automate a translation that into machine configuration (and vice-versa).

[5]The end-to-end principle was originally defined in terms of error correction for communications protocols [SRC84]. We generalize it here, observing that logging of data for forensic purposes is similar to checking data for error-detection purposes.

[6]Likewise, the computer system could be configured to run in an inconsistent and unpredictable way if an attacker was discovered [NB06].

We wish to generalize the formal processes of extracting data from capability models. There are multiple functions algorithms that are involved, depending on the inputs:

The basic function is this: given a requires/provides capability pair describing a single, known action as input, we call the function that outputs the data necessary to log the λ function. The λ function extracts the *difference* in credentials, the *transition* between network locations, the actions that occur, and the *transition* between properties of given services. The λ function *only* operates on a requires/provides pair. We do not define a closed function, but using the procedures we have outlined for each field in the capability 6-tuples above, we assert that it can be done. Informally, we define the λ function as follows:

Let E be a capability pair. Then we define the function λ, which outputs the information to log, as follows:

$$\lambda : E.requires \times E.provides \rightarrow \{(\ldots), \ldots, (\ldots)\}$$

As is shown, the domain is a capability pair, but the range is defined as a set of 8-tuples in the expression below. The ordered 8-tuple is a cross-product and union of the fields in the requires/provides capability pair, with the exception of the *service property* which applies directly and exclusively to the *service.* We describe how this function is processed later in the section in through the use of an algorithm.

$\lambda(Requires, Provides) =$
$\{(A.requires(S.requires.P.requires),$
$A.provides(S.provides.P.provides),$
$C_S.requires,$
$C_S.provides$
$src_S.requires,$
$dest_S.requires,$
$src_S.provides,$
$dest_S.provides),$
$\ldots, (\ldots)\}$

where src_S and $dest_S$ refer to the addresses of the machine running service S, C_S refers to the credentials of service S, and $S.P$ refers to the property P of service S, and $A(S.P)$ refers to an action on service S. The function performs a cross-product on the input pair, so if either the requires set or provides set has more than one 6-tuple, than more than one 8-tuple will result in the output of the λ function.

There are two important processes that we formalize algorithmically. We show the first process in an algorithm, ANALYZE-ATTACK-GRAPH, that describes how to extract information from the attack graph. That algorithm appears in Figure 5.4. The second process, ANALYZE-

Goal, defines how to translate the output of the λ function into an a set of directives on what data to capture for a particular goal, and where to instrument the system to capture that data, since this is not necessarily straightforward. That process appears in Figure 5.6. The algorithm that we use to translate the output of the λ function into a set of directives to implement on a system does not output a machine-specific implementation, but rather uses an English-like intermediate language [BP06] to bridge the gap between the formal model and the real-world analogue. A compilation or translation into a platform-dependent language needs to occur to perform the exact system modifications that it is necessary to make to the system in order to log the appropriate information. A rigorous description of this translation process is beyond the scope in this dissertation, but we present an overview of the process in Section 8.2.

```
ANALYZE-ATTACK-GRAPH ($\Sigma$)
INPUT: $\Sigma$ is a set of $n$ requires/provides capability pairs, $E_1 \ldots E_n$ describing
an attack graph to achieve a specific ultimate goal, $E_n$ and, if $n > 1$, contains
capability pairs for $n-1$ subgoals, $E_1 \ldots E_{n-1}$. Any unknown goals were previously
bounded by using the BOUND-UNKNOWNS algorithm.
OUTPUT: The information necessary to log to analyze the attack
1 for $i = 1$ to $n$ do
2     ANALYZE-GOAL($\lambda(E_i.requires, E_i.provides)$)
```

Figure 5.4: Algorithm for extracting the information necessary to log from an entire attack graph.

The Analyze-Attack-Graph algorithm can also determine the severity of an attack. Severity can be based on a variety of metrics, including the number of attacks possible based on succeeding at a certain sub-goal in the attack graph, or the length of the graph, which may show the difficulty of ultimately achieving the ultimate goal. A final metric that we use looks at the level of access obtained.

In order of severity:

1. no access
2. ordinary user
3. obtaining partial root-level access, such as a file or process that *should* be owned by root (such as `/etc/passwd`), but is not, for some reason; or, convincing a particular process that a user is root, when in actuality they are not (see example in Section 6.6).
4. obtaining root-level access
5. obtaining higher-than-root level access (such as single-user mode)

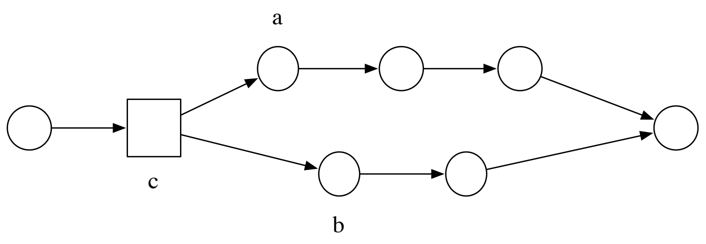


Figure 5.5: An attack graph where circles represent known goals that can be described in advance and squares represent unknown exploits, which cannot.

The purpose of the depth-first-search numbering and traversal of the nodes in the attack graph given as input to Analyze-Attack-Graph is to enable analysis of attack graphs as shown in Figure 5.5. In such a case, both known goals (a,b) on the right side of ab unknown goal (c) must be analyzed individually first, or the unknown goal cannot be analyzed with the τ function.

We now discuss the Analyze-Goal algorithm in Figure 5.6 in detail, which is the algorithm that is applied to the output of line 3 in the Analyze-Attack-Graph algorithm from Figure 5.4:

1. Because each component of a capability pair is a set, there could be more than one 8-tuple in the output of λ, too. This loop addresses that by iterating through each 8-tuple.

2. Each 8-tuple consists of 8 fields of varying types. We address them separately.

3. The first two fields in the 8-tuple are both of type "service.property." Properties are subcategories of services, and since they are tied closely together, are analyzed together.

4. A service alone does not indicate where the "logging point" is. For example, a service could be a shell on a system. If the property were "permission" and the action were "read," the logging point would be the `stat` syscall to check permissions for the use of the "read" syscall. If, on the other hand, the process were memory, the logging point would be wherever memory reads and writes could be observed. Most likely this would be a VM or specialized hardware. Further, a logging point should be as close as possible to the start of the attempt to achieve a certain sub-goal. The closer to the goal that the logging point is, the better the possibility is of analyzing an intruder's attempts to exploit a set of goals. However, automating the localization process is beyond the scope of this dissertation and is something that we will address in the future through the use of policy discovery techniques.

```
ANALYZE-GOAL ($\lambda(E.requires, E.provides)$)
INPUT: $\lambda(E.requires, E.provides)$ is a pair of two sets of one or more 8-tuples,
each, that represents the output of the application of the $\lambda$ function to a capability
pair.
OUTPUT: The information necessary to log to analyze the attack, as real-world
system abstractions
1 for each 8-tuple in the $\lambda$ output (a set)
2       for each field in the tuple
3             if (field = action(service.property))
4                   localize "logging point" by nature of specified action and property
                       of service. Record all specified actions acting on specified service.
5             else if (field = credential)
6                   filter recording of services by recording only those corresponding
                       to elevation in specified credentials:
                       uid, euid, gid, egid, auid of C.requires to C.provides
7             else if (field = address)
8                   record all network communication between specified address(es)
```

Figure 5.6: Algorithm for applying the λ function on a specific sub-goal.

The reasons related to both non-interference and non-deducibility. For example, the Bell-LaPadula model [BL75] says, with regard to multi-level security "No writes down." But what is a "write"? Canonically, it's a file. But it could be to create another entity, which is a covert channel. Though we show a matrix below that indicates a number of possible combinations of action, service, and property, and the resulting logging point, in order to automate the process in the general sense, we would need a specialized policy language and the ability to automate a translation that into machine configuration (and vice-versa).

The following scenarios, on a UNIX system, indicate why action, service, and property are all relevant to the logging point:

action	service	property	logging point
read	program	memory	VM or hardware
allocate	program	memory	kernel (sbrk or mmap syscall)
read	program	permissions	kernel (stat syscall)
read	program	content	kernel (open syscall)
read	net stack	open ports	kernel (socket syscall)
read	net stack	packet data	kernel (read or recv syscall)

As mentioned previously, capabilities and source and destination addresses are important for determining logging requirements, but primarily exist as a filtering mechanism for eliminating unrelated data, and do not help determining the point to log at.

Therefore, the procedures for using our forensic model are as follows:

1. Build the attack graph.
2. Apply the Bound-Unknowns algorithm on the attack graph (which applies the τ and μ functions on the unknown goals in the graph).
3. Apply the Analyze-Attack-Graph algorithm (which in turn calls λ and Analyze-Goal on the individual nodes in the graph).

5.5 Unique Path Identifier

The notion of a *unique path identifier (UPI)* simplifies associating seemingly different segments of attack graphs. A UPI is a tuple that indicates an attacker's position in a graph, associates it with known prior positions, and uniquely identifies all known distinct traversals of a graph. Unfortunately, the markers that would appear to be good UPIs are easily changed on most systems. For example, on a UNIX-like system, a user can change their "effective user ID" (EUID) through legitimate means; therefore, the EUID is not a good unique path identifier.

But combining the identifiers of different processes that are part of the same attack path may be sufficient. Again, on some UNIX-like systems,[7] the “audit user ID” (AUID) is much more difficult to change and so might be a good component of a UPI. The problem is that it disambiguates between *users*, not *paths*. Additional information that could be combined to produce a UPI would be the IP address of the machine they are logging in from (because this is difficult to change), the process identifier, provided child processes maintain a tie to it, and a unique counter to disambiguate multiple occurrences of the same AUID, IP address, and process identifier.

Implementing a unique path identifier when the attack begins would allow data related *only* to the attack to be collected. If an analyst is always able to determine the position of a suspected intruder in the attack graph, then the analyst could collect data about the goals that are bounded, but otherwise *unknown*. The goals that were already *known* could simply be “marked” as having occurred. For the purposes of this dissertation, the data captured from the known goals helps to compensate for the absence of a mechanism to enforce logging strict relative ordering of events.

In a practical implementation of this model, we would like to be able to automate the process of associating an event in a series of intruder’s activities with a particular node in a pre-defined attack graph, as well as associating the event with the events that have preceded it. Our model does this with a unique path identifier. The UPI must contain a set of fields that represent elements of the state of the system, such that when they stay the same, they represent a single, continuous traversal of an attack graph, and when they change, they represent a “break” in a traversal. The UPI must also contain all nodes in all attack graphs that the attacker could currently be at, so that a real-time monitoring system could record where the attacker is currently at in the set of attack graphs, and anticipate the next step(s).

At this point, we cannot prove that a specific set of fields in a UPI enables association in all cases. However, experimental results have shown that the following fields that compose a 5-tuple UPI works in many cases on a UNIX-like system:

1. source IP address of the connection to the system (or localhost),
2. real UID of the user at the point of entry into the attack graph,
3. process ID of the process at the entry of the attack graph,
4. current graph/node positions: $\{(graphID_i, \{goalID_j, \ldots\}), \ldots\}$,
5. unique number

[7]As of this writing, at least Sun Solaris, and possibly others. FreeBSD 6.2 and Darwin (Mac OS X) both implement basic BSM functionality and the BSM API, based on the Common Criteria Guidelines, but the AUID is derived from the “real” UID, and therefore the AUID on those systems is not as immutable as the AUID on Solaris.

The “unique number” represents a known new traversal of the attack graph and merely serves to disambiguate one UPI from any other occurrences of the identical UPI. For example, if more than one occurrence exists of the same IP address, RUID, PID, and attack graph positions, the unique number would increment by one to indicate a new occurrence. The new unique number does not guarantee a new attacker (since it could be the same attacker returning), but a unique number that is the same probably does indicate the same attacker, and therefore the same associated set of events and the same traversal of the attack graph.

The unique number could also be a timestamp of accomplishing the first “goal” in an attack graph. However, because our model does not address concurrent attacks and race conditions, a timestamp would currently be unreliable. Therefore, a unique number serves our purposes.

There certainly exist other fields that would be useful to include in the UPI in other, specific environments. For example, for attacks that involve remote connection via ssh followed by a local exploit, the ssh session ID, which is theoretically unique, could indicate continuity as well.

Though our forensic model contains the concept of a UPI, our examples and experiments do not make use of a UPI, because it is unnecessary in a our limited, proof-of-concept implementations, which do not involve automated association of events.

5.6 Proving the Model

At the moment, we are unable to prove that the output of the model — that is, the data necessary to be logged, and the means of doing so — is sufficient and necessary under all circumstances and for all implementations. We *can* show that the data is sufficient and necessary for *specific* cases. We can show that data is *necessary* by analyzing the effect of removing individual pieces of log data. And, we can show that data is *sufficient* by analyzing the log data and indicating how it helps to understand and analyze an attack. We do both of these things in the next two chapters. In Section 8.2, we discuss methods that future approaches might be able to use to rigorously determine sufficient and necessary data in the general case. Though we can show necessary conditions for specific instances, we cannot currently, show in any formal manner, a more rigorous claim, even only for specific instances: we cannot show that a set of data is *minimal.* This is because there could be ways of analyzing the data to understand an attack that the designers of the attack graph, are unaware of.

5.7 Conclusions

This work presented a forensic model with a set of qualities that enhance the forensic analyzability of the system [PBKM07b] and are derived from a number of principles of forensic analysis [PBKM05].

Returning to the qualities that we described in the previous chapter, we can see that our model meets most of those qualities well:

1. *Ability to log anything*: Our forensic model provides the ability to specify any object or action.

2. *Automated metrics ... and a tuning parameter, that gives a forensic analyst the necessary data and the ability to make the decision as to what to record practically ...*: The measure of severity is based on credentials acquired, and gives some indication to a forensic analyst about what needs to be recorded. By eliminating goals or entire attack graphs that do not concern achieving a certain credential, the severity metric acts as a tuning parameter. Hence, our model has this quality. Chapter 8 discusses planned extensions of this part of our model.

3. *Ability to log data at multiple levels of abstraction, including those that are not explicitly part of system being instrumented*: Our forensic model allows objects and actions ranging from the hardware level to the application level, or even to a non-technological, human process level.

4. *Ability to place bounds on and gather data about portions of previously unknown attacks and attack methods*: The function that analyzes the post-conditions of one attack and the pre-conditions of another attack provides upper bounds on the pre-conditions, and lower bounds on the post-conditions of an event that occurs between the two attacks.

5. *Ability to record information both about the conditions before (cause) and the conditions after (effect) an event has taken place*: Our model is based on the requires/provides model, which easily provides a method to describe both pre-conditions and post-conditions.

6. *Ability to model multi-stage attacks*: The notion of intermediate goals in our model addresses multi-stage attacks.

7. *Ability to translate between logged data and the actual event in a one-to-one fashion*: Chapter 8 describes a method that will add this quality to our model.

In the next chapter, we show examples of applying the model to several different attacks. With each example, we discuss why the information output by the model's algorithms represents a *necessary condition* to analyze the attacks effectively. In Chapter 7, we show the results of implementing several of the examples. We also discuss why the results allow the the attacks to be analyzed, thus demonstrating why the data is also *sufficient* and therefore validating the effectiveness of the model for the implementation of each example.

6

Examples of Using Laocoön

Sherlock Holmes: *"In solving a problem of this sort, the grand thing is to be able to reason backwards. That is a very useful accomplishment, and a very easy one, but people do not practise it much. In the everyday affairs of life it is more useful to reason forward, and so the other comes to be neglected. There are fifty who can reason synthetically for one who can reason analytically."*

—Sir Arthur Conan Doyle, "A Study in Scarlet," *Beeton's Christmas Annual* (1887)

As we discussed before, there are a number of ways in which one could choose a set of attack graphs for use with our forensic model. In Chapter 8, we discuss an automated method of doing this called *policy discovery*. One could also build attack graphs with a methodical penetration testing approach. We chose a number of examples to model using our methodologies because they cover all classes of flaw domains as enumerated in the seminal RISOS [ACD$^+$76] and Protection Analysis (PA) [BH78] reports. It is possible that a large enough collection of examples, with enough overlapping coverage of the flaw domains, might be sufficient to analyze not just the attacks specified but any other attack in the same flaw domain(s) as the specified attacks. At this time, we do not assert this, but instead use the fact that these examples cover all flaw domains in the two reports as a means of demonstrating the effectiveness of the model. That is, we believe our model should be effective for most situations covered by these flaw domains.

The following is a list of the RISOS flaw domains:

RISOS #1: Incomplete parameter validation
RISOS #2: Inconsistent parameter validation
RISOS #3: Implicit sharing of privileged/confidential data
RISOS #4: Asynchronous-validation/Inadequate-serialization
RISOS #5: Inadequate identification/authorization/authentication
RISOS #6: Violable prohibition/limit

RISOS #7: Exploitable logic error

The following is a list of the PA flaw domains:[1]

PA #1a: Improper choice of initial protection domain

PA #1b: Improper isolation of implementation detail (exposed representations)

PA #1c: Improper change (consistency of data over time)

PA #1d: Improper naming

PA #1e: Improper de-allocation or deletion (residuals)

PA #2: Improper validation (of operands, queue management dependencies)

PA #3a/b: Improper synchronization (indivisibility + sequencing)

PA #4: Improper choice of operand or operation (critical operation selection errors)

The following list indicates the scenarios and exploits that we discuss in detail and the sections of this dissertation that describe their application, and, in some cases also their implementation:

- buffer overflow (sections 6.1 and 7.1)
- spyware (sections 6.2 and 7.2)
- ignoring permissions (sections 6.3 and 7.3)
- omitting or ignoring authentication (sections 6.4 and 7.4)
- Trojan horse (sections 6.5 and 7.5)
- bypassing standard interfaces (Section 6.6 and 7.6)
- inconsistent parameter validation (Section 6.7)
- LAND attack (Section 6.8)
- shared memory code injection (Section 6.9)
- 1988 Internet worm (Section 6.10)
- Christma Exec worm (Section 6.11)
- NFS exploits (Section 6.12)

The following list indicates each exploit and the flaw domain that it covers:

[1]The numbering scheme that we use for the PA flaws corresponds to the revised hierarchy outlined by Neumann [Neu78], not the numbering in the original PA paper.

	PA								RISOS						
exploit	1a	1b	1c	1d	1e	2	3a/b	4	1	2	3	4	5	6	7
buffer overflow		x	x			x	x		x					x	
spyware	x										x				
ignoring permissions					x		x					x	x		
authentication						x		x					x		
Trojan horse				x											x
bypassing interfaces	x	x	x								x				
parameter validation						x				x					
LAND attack						x		x			x		x		
shmem injection	x	x	x			x	x					x			
1988 Internet worm	x	x	x	x	x	x		x	x	x		x	x	x	x
Christma Exec worm	x			x							x				
NFS expoits	x	x				x		x			x		x	x	

In this chapter and the next chapter, we show several examples of applying and implementing our forensic model on a UNIX-based system. Given that at this point we lack automated policy discovery techniques, we make the simplifying assumption that we have the attack graphs for the policy violations associated with each attack, and that those attack graphs are complete.

Because we cannot guarantee that every path — particularly those that involve covert channels — is accounted for, we currently do not indicate covert channels in our attack graphs. Similarly, because we assume that the attack graphs are complete, we also do not indicate covert channels in our discussion of the data that the model indicates must be logged. However, there is no reason why covert channels could not be indicated and analyzed in the future, given a more rigorous, automated method of generating attack graphs based on policy violations.

Finally, though we show that the information required by the model to be recorded is sufficient to analyze each exploit, other information, such as time of day or full paths of executables, frequently might be useful additional, contextual information. However, when that information is not explicitly required by the model, we do not indicate it in our discussion of the results of applying the model.

6.1 Obtaining a Root Shell

We apply our forensic model to an intruder obtaining a *root* shell on a UNIX-like system. Buffer overflow exploits are commonly used to gain a UNIX shell, especially when the software being attacked is run as root. Even software that provides security functionality has been compromised in this way [Com99a, Com99b, Com00, Com01, MSB$^+$06]. Some of the most common remote exploits have been against the SMTP mail agent `sendmail`, web servers, and programs associated with web servers, such as CGI scripts. While most web servers do not run as root, some CGI scripts must and do run as root (for example, a script to change a password over the web). These may also contain buffer overflow vulnerabilities [Bis86]. Therefore, we consider an exploit of a web server to gain a root shell as an example of our methodology.

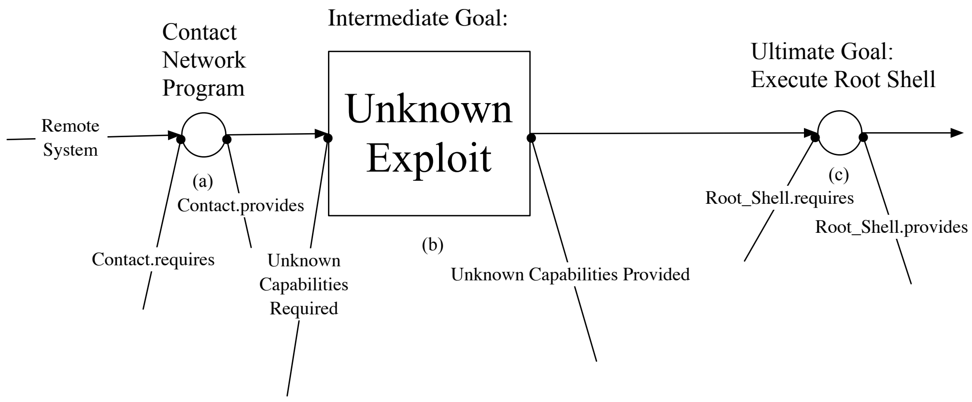


Figure 6.1: Diagram of a remote attack, exploiting a network program to obtain a root shell. (a) represents the remote connection. (b) represents the exploit that occurs to obtain the shell. In an experiment that we show later, this is a buffer overflow, but it could be many different things. Hence, we do not model this directly. (c) represents executing the root shell.

Consider a web server as the victim of a remote buffer overflow [Ale96] exploit of a CGI script on a web server to gain root privileges of a web server. We model the process of obtaining root privileges remotely without limiting ourselves to any particular exploit. One starting point—network contact—is common to all such attacks. This could be a full TCP connection, a TCP SYN packet, or a UDP packet. The buffer overflow could be a traditional overrun of an unchecked buffer on the stack, an integer overflow, format string vulnerability, heap overflow, off-by-one error, or function pointer overwrite. The exploit may not be a buffer overrun at all. So we capture as many possible paths from the remote contact to the execution of the root shell.

The end goal is the execution of the root shell. The first goal is contact with a network-aware program. At least one intermediate step is the exploit of the network-aware program. A second intermediate step is manipulating a program to the point of issuing arbitrary commands. This could be part of the exploit itself (for example, if the program is `setuid root`,[2] or it could be a distinct step (for example, if the program executes a second program that is `setuid root`).

[2]Exploits of setuid root programs are among the most common local exploits to gain superuser privileges [Bis86].

The first capability set shows the capabilities before the network contact, and the second shows the capabilities after the network contact.

Remote_Connect.requires:
$\{(ANY(IP), local, \emptyset, Communicate.recv, Net\_Process : \mathbf{P}, code)\}$
where $Net\_Process$ is a process that has an open network port.

This means a user must be able to connect to the machine over the network and that an exploitable program is listening on an open network port. The connection is not the exploit; it is just a socket listen/connect. The exploit comes later.

Remote_Connect.provides: $\{(ANY(IP), local, \emptyset, Net\_Process : \mathbf{P}, \emptyset, \emptyset)\}$

The program is exploited after writing something unexpected to the open socket. After manipulating the network program, the intruder is considered "local" and can perform any action consistent with the user who owns the exploited program. This might be user *daemon* or it might be *root*. But we can provide a capability pair for the exploit itself only if we know precisely what the exploit is. If we do not know what the exploit is, we certainly cannot describe the conditions that will permit it to happen, nor can we precisely describe the conditions that it provides.

We now consider the local scenario:

Root_Shell_Local.requires: $\{(local, local, \mathbf{C'}, exec, Process : (\mathbf{P} \wedge \mathbf{S'}), code)\}$
where $\mathbf{P}$ refers either to the process in *Remote_Connect.requires* or another one if the exploit begins and ends locally, and
if $\mathbf{C'} = \{(euid = root \vee uid = root)\}$ then $\mathbf{S'} = ANY(shell)$,
else, if $\mathbf{C'} = \{uid.exists = true \wedge (euid \neq \texttt{root} \wedge uid \neq \texttt{root})\}$,
then $\mathbf{S'} = setuid\_root\_shell$.

If the network program exploited was setuid root, then the intruder can use any shell. If the network program exploited was run by a valid user but was not setuid root, the intruder needs a setuid root shell. The *same* model can be used if a user *already* has a user account on a system.

Root_Shell_Local.provides:
$\{(local, local, euid = \texttt{root} \vee uid = \texttt{root}, ANY(Action), shell, ANY(Property))\}$

We apply the Analyze-Attack-Graph algorithm discussed earlier to determine the information to log based on the requires/provides pairs. We apply the algorithm from the ultimate goal — the endpoint of the attack — and work towards the beginning.

Applying the λ function, per line 2 of Analyze-Attack-Graph, we see:

$\lambda(\mathit{Root\_Shell\_Local.requires}, \mathit{Root\_Shell\_Local.provides}) =$
$\{(\mathit{exec}(\mathit{Process} : (\mathbf{P}.\mathit{code} \wedge \mathbf{S'}.\mathit{code})), \mathit{ANY}(\mathit{Action})(\mathit{shell.ANY}(\mathit{property})),$
$\mathit{euid} = \mathbf{C'}, \mathit{euid} = \texttt{root}, \mathit{local}, \mathit{local}, \mathit{local}, \mathit{local})\}$

The ultimate goal of the attack is the local *exec* of a shell with a uid or effective uid of *root*. Applying the Analyze-Goal algorithm to the *Root_Shell_Local* goal, the model indicates that the process to record *sufficient information* to analyze this particular goal is:

1. action(service.property): Localize the logging point — the point at which to monitor and log the information — at the `exec` system call (first field in the λ output). Record the transition from "any" program (**P**) to a shell (**S'**) to any other programs that are spawned from the shell (second field in the λ output), by recording executions and any programs spawned after the executions. This also includes the path of the program executed.

2. transition of credentials: The transition is from any local user (**C'**) to uid or euid = root (third and fourth fields in the λ output).

3. transition between src/dest network addresses: None, as all are local (fifth through eighth fields in the λ output).

To summarize, *all* shell executions and other programs invoked with a uid or euid of *root* are captured. This will not capture the execution of a non-root shell. If that is desired, the goal must be made more general. While recording this information will not guarantee that the action is malicious, it should avoid most false negatives. Further, this analysis only covers the exploit itself and not what occurs after the exploit. The information to be logged after the exploit should be guided by additional attack graphs and/or the security policy of the site. Such policies might state that after an intruder obtains root access, actions in addition to program executions must be logged, perhaps down to the detail of memory reads and writes.

The next step in the attack is the exploit, which we do not know anything about. But we do know something about the events before (the remote connection) and after (obtaining the root shell). Therefore, we apply the Bound-Unknowns algorithm (which in turn applies the τ function) to determine the information to log, as follows:

$$\tau(\mathit{Remote\_Connect.provides}, \mathit{Root\_Shell\_Local.requires})$$
$$= (\mathit{Exploit}_{max}.\mathit{requires}, \mathit{Exploit}_{min}.\mathit{provides})$$

Before we analyze the *Exploit* goal further, we apply the Analyze-Goal algorithm to the goal that precedes it — the *Remote_Connect* goal.

$\lambda(Remote\_Connect.requires, Remote\_Connect.provides) =$
$\{(Communicate.recv(Net\_Process : \mathbf{P}.code), Net\_Process : \mathbf{P}, \emptyset, \emptyset,$
$ANY(IP), ANY(IP), local, local)\}$
where $Net(Process)$ is a function that describes a process that has an open network port.

The algorithm indicates that the process to record *sufficient information* to analyze the goal is as follows:

1. action(service.property): Localize the logging point as the socket system calls involved in receiving communication (`listen, receive`), made during the running of ***P****.code* (first field in the λ output). Also record the name and path of the program making the calls. The action of receiving the communication begins the attack.
2. transition of credentials: None. Local login has not been achieved (third and fourth fields of the λ output).
3. transition between src/dest network addresses: In this case, the IP addresses of any machine connecting to the local machine (fifth and sixth fields of the λ output, respectively), including the local machine sending packets to itself.

Now, apply Analyze-Goal to the capability pair of the *Exploit* goal:

$Exploit_{max}.requires$:
$\{(ANY(IP), local, \emptyset, Net\_Process : \mathbf{P}, \emptyset, \emptyset)\}$
$Exploit_{min}.provides$: $\{(local, local, \mathbf{C'}, exec, Net\_Process : (\mathbf{P} \wedge \mathbf{S'}), code)\}$ where
if $\mathbf{C'} = \{(euid = root \vee uid = root)\}$ then $\mathbf{S'} = shell$,
else, if $\mathbf{C'} = \{uid.exists = true \wedge (euid \neq \texttt{root} \wedge uid \neq \texttt{root})\}$, then $\mathbf{S'} = setuid\_root\_shell$.
$\lambda(Exploit_{max}.requires, Exploit_{min}.provides)$:
$\{(Net\_Process : \mathbf{P}, exec(Net\_Process : \mathbf{P}.code), \emptyset, euid = root, ANY(IP), local, local, local,)\}$

The algorithm indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point as the `exec` system call (second field of the λ output). Record all *exec*s of or by ***P*** that result in an effective uid of root (second and fourth fields of the λ output). Also, record the names of the service(s) that provide the ability to execute a root shell.

2. transition of credentials: The euid/uid/gid of program **P** (previous goal) and any programs that are spawned root from the program **P** (fourth field of the λ output.

3. transition between src/dest network addresses: Communication between a remote source and the local machine (fifth through eighth fields of the λ output).

Summarizing, the *sufficient information* needed to analyze an intrusion of a network program to obtain a root shell is:

1. Initially: Record all communication into the machine that is received (via socket `receive` system call) by programs with open network ports. The system should add or delete programs from a list as they invoke the socket `listen` syscall.

2. After connection with the of the network program: Record all `exec`s and record all changes to the `uid` and `euid` of a user and the program in which it happens.

3. After the attacker has acquired a root shell: Record all syscalls and all context.

To see whether this series of recorded data always represents a remote exploit of a setuid root program to gain access to a shell, consider what the absence or presence of only a subset of the elements recorded in this attack graph indicates:

- Examining the *Remote_Connect* goal alone cannot detect this attack. That goal represents communication. It could be a user contacting a web server.

- Also, examining the *Root_Shell_Local* goal alone indicates a successful execution of a shell with euid root. This may not be a *bad* thing, if, for example, it is the result of a successful `su` or `sudo` (a later example analyzes the difference between proper and improper elevation of credentials). However, this gives us a place to begin the analysis for how the intrusion proceeded since we can tell which program called the `exec` that started the shell. The data from the *Remote_Connect* goal tells us if the intrusion was external.

- The *Exploit* goal might consist of many different types of exploits and many different events unrelated to the exploit, such as forks and execs that occur between the connection and the exec of the root shell. Examining his goal teaches nothing about either the origin of the attacker or the result of the attack. Rather, it adds data to that for the *Remote_Connect* and *Root_Shell_Local* goals.

Therefore, the information specified by the model represents a *necessary condition* to be able to analyze this particular exploit.

6.1.1 Arbitrary Commands

One problem with the models above is that from a logging perspective, we cannot necessarily describe in advance what a "shell" is. The concept of a shell is not universal. The main point is actually that the user can execute arbitrary commands. There is nothing about a shell, other than convenience, that allows a user greater privileges than any other program. In other words, a root shell is one mechanism by which an intruder could execute arbitrary commands, but not the only one. The following descriptions characterize this for a local intrusion:

$Allow\_Arbitrary\_Commands.requires$: $\{(local, local, \mathbf{C'}, exec, Program : \mathbf{P}, code)\}$ where
$\mathbf{C'} = \{((euid \neq \texttt{root} \wedge uid \neq \texttt{root}) \wedge user.exists = true)\}$

$Allow\_Arbitrary\_Commands.provides$:
$\{(local, local, (euid = \texttt{root} \vee uid = \texttt{root}), exec, ANY(Program), code)\}$

$\lambda(Allow\_Arbitrary\_Commands.requires, Allow\_Arbitrary\_Commands.provides)$:
$\{(exec(Program : \mathbf{P}.code), exec(ANY(Program).code), \mathbf{C'}, euid = root,$
$local, local, local, local,)\}$

In other words, what we really want to know is the transition from a status of a normal user to that of a superuser, and the programs (and permissions/scenario) that allowed this to happen.

For *Allow_Arbitrary_Commands*, the ANALYZE-GOAL(λ) function indicates that the process to record *sufficient information* to record is:

1. action(service.property): Localize the logging point again at the `exec` system call from any program to any services that follow, by recording executions and any actions that occur after the executions.

2. transition of credentials: From any existing, local credentials, to root.

3. transition between src/dest network addresses: None; all are local.

So the summary of all three goals: *Remote_Connect*, *Exploit*, and
Allow_Arbitrary_Commands, is:

1. Initially: Record all communication into the machine that is received (via socket `receive`) by programs with open network ports (a list of these should be maintained and added/subtracted to/from as programs indicate `socket.listen`).

2. After connection with the of the network program: Record all `exec`s and record all changes to the `uid` and `euid` of a user and the program in which it happens.

3. After the attacker has acquired a *program* as root: Record all syscalls and all context.

6.2 Spyware (e.g. a Trojaned sshd)

Consider an sshd daemon that has been Trojaned to capture user passwords and send them to another machine over the network. Such a compromise occurred at the San Diego Supercomputer Center (SDSC) in 2004 [MB05, Sin05]. Many such examples of Trojaned security software exist [Com99c, Tho84]. The result was over a year of largely unknown and unrecorded intrusions by a 15-year old "hacker" in Sweden.

The data collected by the systems at SDSC were exclusively related to syslog entries indicating logins to UNIX machines via `ssh`, as well as `utmp` and `lastlog` data that indicated the terminals connected to and the durations of the connections. While ultimately this was the data that indicated that a password had probably been captured (because a user noticed that a login to her account had occurred at a time and from a location that was unfamiliar to her), this realization took so long that the attacker was able to abuse multiple accounts for months, unobserved. Also, the logs still did not say what the attacker actually *did*.

The ultimate goal of this attack is to send the passwords over the network. This is different than the *implication* of the attack, as we discussed earlier. The implication is that the password can be used by the intruder to log back into the system in the future and view or modify the system, as permitted by the credentials of the user whose account is compromised. But that implication is not the ultimate goal of the attack that we model. We want to know how the password was compromised in the short-term, while the long-term implications can be analyzed separately. Therefore, the intermediate goal is capturing and storing the password. The ultimate goal is enabled by possessing the password and the ability to send it over a network. Figure 6.2 shows the attack graph.

Applying the Analyze-Attack-Graph algorithm to the attack graph, we begin with the first goal, *Send_Password*:

Send_Password.requires:
$\{(local, ANY(IP), ANY(uid), Communicate.send, ANY(Program : \mathbf{P2}), password : \mathbf{P})\}$

This simply provides the password to someone else, which also allows the ability to re-connect to the localhost. Thus, we need to look for connections back to the account of which the password was compromised.

Send_Password.provides: $\{(ANY(IP), local, \emptyset, Communicate.connect, login(\mathbf{P}), Account(\mathbf{P}))\}$

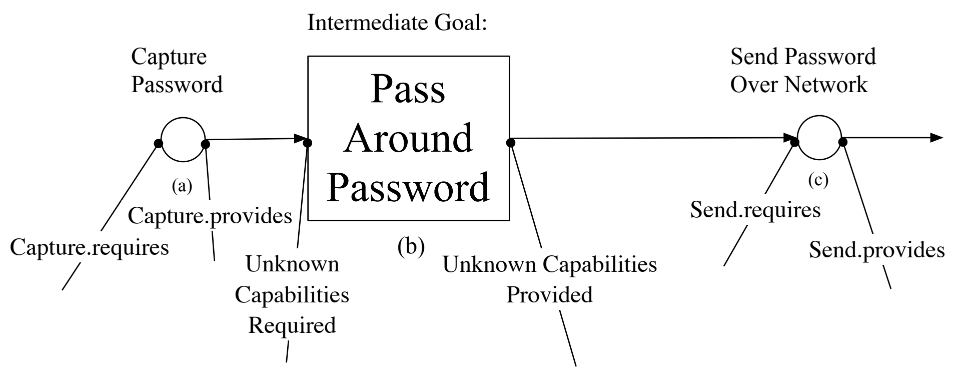


Figure 6.2: Diagram of spyware capturing a password and sending it over a network. (a) represents the capturing of the password. (b) represents sending the password around the machine to another program. We do not know the mechanism used to do this, hence, we do not model this directly. (c) represents sending the password over the network.

To determine the information to log, we apply our algorithm from the ultimate goal (the endpoint of the attack) and work towards the beginning.

Per line 1 of Analyze-Attack-Graph, we start applying the algorithm from the ultimate goal, the endpoint of the attack, and work towards the beginning.

$\lambda(Send\_Password.requires, Send\_Password.provides)$:
$\{(Communicate.send(ANY(Program).password : \mathbf{P}),$
$Communicate.connect(login(\mathbf{P}).Account(\mathbf{P})),$
$ANY(uid), password : \mathbf{P}, local, ANY(IP), ANY(IP), local)\}$

The first step is to apply Analyze-Goal to λ(Send_Password), as the ultimate goal is *Send_Password.* The output indicates that the process to record *sufficient information* to record is:

1. action(service.property): Localize the two logging points as the `send` and `connect` system calls, respectively, of the program **P2** sending out a suspected password **P**, and any logins to the account with the compromised password.

2. transition of credentials: The user ID **u** of the program sending the password.

3. transition between src/dest network addresses: From a remote IP to local.

So the algorithm indicates that the *sufficient information* to analyze the attack consists of the name and path of the program used to send the password, the owner of the program, the location the password is sent to, and the location(s) used to connect back using the password.

Unfortunately, the program used to capture the password may not be the same program used to send the password. Further, the password may be sent over a covert channel, or may be disguised using encryption or other means. We focus on events we can confirm, ignoring covert channels. Therefore, we examine the following areas, reflecting the attack graph, in order to determine what to log and where to instrument the system in order to log the required information:

1. instances when a password *could* be captured (e.g., the systems `read` system call, or anything calling or observing the call being made),
2. instances of known ways in which the password is sent (e.g., the `send` and `connect` system calls), and
3. all instances of when a password is used (e.g., logging back in), following being captured.

Any program that requests a password can capture it.[3] Programs and functions that do so, such as `sshd` and `su`, call or are called by `pam_authenticate`,[4] or any program that spawned the program that calls `pam_authenticate`. Therefore, we capture all invocations of the `pam_authenticate` function.

Capture_Password.requires:
$\{(local, local, ANY(uid : \mathbf{u}), read, ANY(\mathbf{PA}) \wedge ANY(shell) \wedge kernel, password : \mathbf{P})\}$
where **PA** is a program calling `pam_authenticate`

Capture_Password.provides: $\{(local, local, uid : \mathbf{u}, know, Account(\mathbf{P}), password(\mathbf{P}))\}$

Again, per the Analyze-Attack-Graph algorithm, we apply Analyze-Goal to the *Capture_Password* goal:

$\lambda(Capture\_Password)$:
$\{(read(ANY(\mathbf{PA}).password : \mathbf{P} \wedge ANY(shell).password : \mathbf{P} \wedge kernel.password : \mathbf{P}),$
$\emptyset(Account(\mathbf{P}).password(\mathbf{P})), ANY(uid), ANY(uid),$
$local, local, local, local)\}$
where **PA** is a program calling `pam_authenticate`

The model indicates that the process to record *sufficient information* to record is:

1. action(service.property): Localize the logging point in the `read` system call.

[3]For our example, we assume that users will enter their passwords only when a trusted program requests it.

[4]PAM is the "pluggable authentication module" used on FreeBSD, Solaris, and a number of other UNIX-like operating systems.

2. transition of credentials: The user ID **u** of the program capturing the password.
3. transition between src/dest network addresses: None; all are local.

As with the process of obtaining a root shell, this attack graph involves a known beginning and ultimate goal, and an unknown intermediate goal between the two. The intermediate goal is the possible communication of the password between the program that captures the password and the program that sends it over the network. So we apply the BOUND-UNKNOWNS algorithm and the τ function:

$$\tau(\mathit{Capture\_Password.provides}, \mathit{Send\_Password.requires})$$
$$= (\mathit{Intermediate}_{max}.\mathit{requires}, \mathit{Intermediate}_{min}.\mathit{provides})$$

$\lambda(\mathit{Intermediate}_{max}.\mathit{requires}, \mathit{Intermediate}_{min}.\mathit{provides})$
$= \{(\emptyset(\mathit{Account}(\mathbf{P}).\mathit{password}(\mathbf{P})), \mathit{Communicate.connect}(\mathit{ANY}(\mathit{Program}).\mathit{password} : \mathbf{P}),$
$\mathit{local}, \mathit{local}, \mathit{local}, \mathit{ANY}(\mathit{IP}))\}$

Analyzing this pair with ANALYZE-GOAL($\lambda(\mathit{Intermediate}_{max}.\mathit{requires},$ $\mathit{Intermediate}_{min}.\mathit{provides})$ indicates that the process to record *sufficient information* to record is:

1. action(service.property): Localize the logging point in the mechanisms that can communicate the data between the two processes **PA** and **P2**: read/write shared memory, IPCs, read/write disk, and send/receive over the network.
2. transition of credentials: The user ID **u**.
3. transition between src/dest network addresses: The password has not yet been sent over the network, so all communication is local at this time.

Summarizing, the model indicates that the following information is *sufficient* to analyze the attack:

1. Any calls to `pam_authenticate`, the return value, and the program and user who invoked it.
2. The IP address of any connection from the program in question, the shell used at the time, or the kernel. (Could record any open socket, but this may result in many false positives. This is a possible tuning parameter.)

3. The IP address of any connection using the password previously typed.

4. Possible communication within the machine.

Collecting the IP address of every connection might involve recording more information than is practical given limited disk space. An alternative is to link the password capture step with the password send step. This involves monitoring any information sent between a program that ordinarily calls `pam_authenticate` and any program that makes network transmissions. While this involves recording less information, it also involves enumerating in advance all the programs that ordinarily call `pam_authenticate`. Maintaining such a list would be difficult if a system has programs added or removed frequently.

Failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. For example, failing to record the program (and surrounding details) that captured the password fails to indicate the source of the capture, and failing to record the program making the outgoing socket connection fails to record the destination. Finally, failing to record the IPC between those two programs hinders an analyst's ability to associate the two events. Therefore, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.3 Modify /etc/passwd (e.g. via lpr bug)

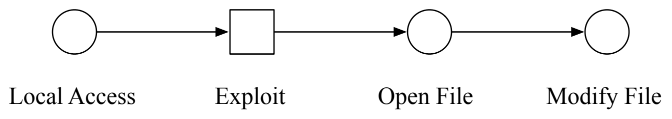


Figure 6.3: Diagram of re-writing a privileged file by exploiting multiple bugs in the UNIX program `lpr`.

A bug in `lpr` [8LG] involving re-use of spool files and a non-atomic `open` command allows an attacker to replace any file, including `/etc/passwd`. The exploit is a classic time-of-check-to-time-of-use race condition [BD96]. The privileged program first checks permissions of a file and determines if it can be altered. If so, before the file can be opened for writing, the attacker replaces the file with a symbolic link to the target file. As the program runs with root privileges, it opens the file for writing — and therefore overwrites the file that the link points to. The broader issue is simply manipulating bugs in a `setuid root` program to modify `/etc/passwd` or `/etc/groups`. A high-level attack graph appears in Figure 6.3. Note that this does not specifically indicate the multiple vulnerabilities exploited during a traversal of this graph, but instead looks at the attack more generally, so that the same attack graph might be more easily applied to other, similar attacks.

The circumstances require the ability to write `/etc/passwd`, which means that the program used must have root privileges. This capability can include any file that is owned by root; for simplicity, we focus on `/etc/passwd`.

The capability set for the requires portion of the first goal in the attack graph is as follows:

LocalAccess.requires: $\{(local, local, uid = ANY(user : u), \emptyset, \emptyset, \emptyset)\}$

The capability set of the provides portion is identical, simply indicating that local access must be first obtained before any possible exploit.

LocalAccess.provides: $\{(local, local, uid = ANY(user : u), \emptyset, \emptyset, \emptyset)\}$

The model indicates that the process to record *sufficient information* to record is:

1. action(service.property): None; no specific actions and services.
2. transition of credentials: None; no change in uid. However, it remains constant at uid:**u**, which we need to record.
3. transition between src/dest network addresses: None.

The capability pair for the goal of opening the `passwd` file is as follows:

Open_passwd.requires:
$\{(local, local, uid = ANY(user : u) \wedge euid = root, Open, file, /etc/passwd)\}$
Open_passwd.provides: $\{(local, local, euid = root, Write, file, /etc/passwd)\}$

The model indicates that the process to record *sufficient information* to record is:

1. action(service.property): Localize the logging point in the kernel, in the `open` system call (that provides the file handle) and the `write` system call (that takes the file handle and writes or changes the data), which are the means of writing files.
2. transition of credentials: From uid:**u**, to an effective uid of root.
3. transition between src/dest network addresses: None; all actions are local.

As with the previous examples that we have shown, we apply the BOUND-UNKNOWNS algorithm and the τ function to analyze the intermediate step; that is, the exploit:

$$\tau(Local\_Access.provides, Open\_passwd.requires)$$

$$= (Exploit_{max}.requires, Exploit_{min}.provides)$$

$\lambda(Exploit_{max}.requires, Exploit_{min}.provides)$
$= \{(Write(file./\texttt{etc/passwd}), Write(file./\texttt{etc/passwd}), euid = root, euid = root$
$local, local, local, local)\}$

Analyzing this pair with ANALYZE-GOAL($\lambda(Exploit_{max}.requires, Exploit_{min}.provides)$ indicates that the process to record *sufficient information* to record is:

1. action(service.property): Localize the logging point in the kernel, in the `write` system call (that takes the file handle and writes or changes the data).

2. transition of credentials: None, but we still record the actions of the uid of the user and the status of whether the uid or euid is root, to determine when and how privileges are escalated.

3. transition between src/dest network addresses: None; all actions are local.

Finally, the capability pair for actually modifying the `passwd` file is:

Modify_passwd.requires: $\{(local, local, euid = root, Write, file, /etc/passwd)\}$

Modify_passwd.provides: $\{$
$(local, local, user : \mathbf{u}, know, ALL(users), usernames)$,
$(local, local, user : \mathbf{u}, Write, ALL(users), Accounts)\}$

This gives the ability to write that file, knowledge of the usernames, and finally, by extension, knowledge of all other passwords (because they can either be decrypted or replacedwith whatever the attacker desires).

The model indicates that the process to record *sufficient information* to record is:

1. action(service.property): Localize the logging point in the kernel, in the `write` system call (that takes the file handle and writes or changes the data).

2. transition of credentials: None, but we still record the actions of the uid of the user and the status of whether the uid or euid is root, to determine when and how privileges are escalated.

3. transition between src/dest network addresses: None; all actions are local.

To summarize the information to record to analyze the entire attack: Record all root-owned files that are written (this includes appending, moving, copying, and deleting), the program that does so, and the user who has executed the program. This includes files that have been linked to (and therefore links and symlinks must also be recorded), or files that have been written to by bypassing the file system and writing directly to the raw device.

Failing to record the symbolic links (and surrounding details) that indicate the actual file opening fail to indicate the actual file being opened. Additionally, failing to record the permissions of the program making the calls to `open` and `symlink` fails to record the process by which `lpr` is manipulated to replace the `/etc/passwd` file. Therefore, again, failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack, and thus, the output of the model also represents a *necessary condition*.

6.4 Avoid Authentication (e.g. in su)

If a setuid root program that results in a root shell (e.g. `su`) is modified to not call the `pam_authenticate` routine, or to ignore the results, and always consider the user's password valid, `root` access can be achieved without knowing the root password.

How do we generalize a model of this so it could apply not only to `su` but also to `ssh` or `rlogin`, for example? An attacker requires the ability to call `pam_authenticate` without success *or* to avoid `pam_authenticate` entirely. The capability pair is as follows:

$Avoid\_Auth.requires$:
$\{(local, local, ANY(user), call, \texttt{pam\_authenticate} \neq success \vee \texttt{pam\_authenticate}.exists =$
$false,$
$code)\}$

$Avoid\_Auth.provides$: $\{(local, local, ALL(users) \wedge euid = root, exec, shell, ANY(property))\}$

Therefore, the result of applying the Analyze-Attack-Graph algorithm is as follows:

$\lambda(Avoid\_Auth)$:
$\{(call(\texttt{pam\_authenticate}.code), exec(shell), ANY(user), ALL(users),$
$local, local, local, local)\}$

The key to this is to determine whether a user's pre-authentication credential and post-authentication credential has changed, and, if so, whether that was because `pam_authenticate` was successful or bypassed. Therefore, the model indicates that the process to record *sufficient information* to record in order to analyze the attack is:

1. action(service.property): Localize the logging points to record the `pam_authenticate` library call either by instrumenting the call itself, or using a tool like `ltrace` to trace dynamic library calls. A second logging point is the `exec` system call, which covers the recording of shell executions from the kernel.

2. transition of credentials: From one user **u** to any other user, including root.

3. transition between src/dest network addresses: None; all actions are local.

To summarize:

1. Record all invocations of `pam_authenticate`, including the arguments (and password) given. Record the program from which it is called and the uid/euid of the user before the call, so the vulnerable program can later be identified and the child process of the vulnerable program can be positively identified by its relation to the PID of the parent/vulnerable program.

2. Then determine the uid/euid of the program *after* the call, and any programs (shells, most likely) which are *exec*-ed following the termination of the program containing `pam_authenticate`.

3. Check with the system to determine if the user really was successful. One way of doing this is by replaying the use of the logged/captured passwords to determine whether they are a matched pair. If not, note this.

4. If the uid/euid of a user has been elevated *without* calling `pam_authenticate` at all, note this.

One flaw with this method is that we have to assume that `pam_authenticate` and the other related routines in PAM have not also been modified to return a "success" result. However, if we do not make these assumptions, an equally reasonable alternative to monitoring `pam_authenticate` is also to monitor the `open` system calls for whether `/etc/spwd.db` has been opened for reading.

The `pam_authenticate` call is responsible for the system's password-checking process. In this simple example, failing to record it, and the surrounding details, leaves an analyst with few ways of understanding when a password might actually be captured but ignored. Therefore, again, failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. Thus, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.5 Trojan Horse (e.g. via search path modification)

In an early version of UNIX, there was a flaw in `vi`[5] such that when an editing session was terminated unexpectedly, it would mail the user about how to recover data from the lost session, not with a full path (e.g. `/bin/mail`, but with simply a program: `mail`). As a result, if a binary called `mail` were placed higher in the search path order than `/bin`, the binary, which could be a Trojan horse [And72], would be executed. Further, since `vi` used a privileged program to store temporary files on the system, the Trojaned version of `mail` would be executed with superuser privileges.

Without enhanced logging, the events are invisible to the user, and given that they do not require entering any special `su` or `sudo` commands, also are invisible to a system administrator. Even with process accounting turned on, the Trojaned program looks just like any other.

Our experiment involves a very minimal program:

```
int main() {
        FILE * F;
        F = popen("date","r");
        printf("Opened\n");
        pclose(F);
}
```

In this program, the `date` program is called with `popen`, similar to the original bug in `vi`. The circumstances are such that the path is set to:

```
.:/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin
```

The system's `date` program is located in `/bin`. In the first case, that is the only `date` in the path. In the second case, a "rogue" `date` program is also in the current working directory. In reality, it could be easier to modify a user's path without their knowledge, or to put a malicious program higher in their "search path" than the system's, than to obtain `root` access. This is because inserting a malicious program only requires access to a *user's* account, not necessarily `root`. Therefore, any number of password-sniffing, social engineering, or other

[5]See p. 90, slide #171 in *How Attackers Break Programs, and How to Write Programs More Securely* [Bis02] at `http://nob.cs.ucdavis.edu/bishop/secprog/sans2002.pdf`

ordinary techniques might enable an attacker to do this. We can generally analyze modification of the search path to gain root privileges by recording the search path used for execution of programs.

The actions require: (1) a login, (2) the ability to read another user's search path, (3) a directory writable by the attacker in the search path before the relevant system directory; or (2) the ability to reset the search path, and (3) as before. The capability pairs are as follows:

$Trojan\_Horse.requires$:
$\{(local, local, ANY(user : \mathbf{u}), modify, Account(\mathbf{u}), execution\_search\_path),$
$(local, local, uid(\mathbf{u}) \wedge euid = root, exec, program(\texttt{Trojan}), code)\}$

The result gives the ability to execute any program as `root`.

$Trojan\_Horse.provides$: $\{(local, local, ALL(users) \wedge euid = root, exec, program, code)\}$

Therefore, the result of applying the Analyze-Attack-Graph algorithm are as follows:

$\lambda(Trojan\_Horse.requires, Trojan\_Horse.provides)$:
$\{(modify(Account(\mathbf{u}).execution\_search\_path), exec(program.code), uid : (\mathbf{u}), euid = root,$
$local, local, local, local),$
$(exec(program(\texttt{Trojan}).code), exec(program.code), uid : (\mathbf{u}), euid = root,$
$local, local, local, local)\}$

The model then indicates that the process to record *sufficient information* to analyze the attack is:

1. action(service.property): Localize the logging point as the `exec` system call, also additionally collecting the execution search path. Both can be done at the same time upon execution (which is the only time that the execution search path matters).

2. transition of credentials: From any user **u** to euid=root.

3. transition between src/dest network addresses: None, as all are local.

Therefore, all changes of effective UID must be recorded. Failing to record all of the information indicated results in either an incomplete list of executions, or an incomplete specification of the program actually executed. Thus, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.6 Bypassing Standard Interfaces (e.g. via utmp bug)

An exploit in 1994 involved a world-writable `/etc/utmp` [Bis83, Com94b]. The `/etc/utmp` file is a binary representation of all currently logged-in users. Each user corresponds to an entry in the form of a data structure with four fields: terminal name, user login name, originating host, and the time that the user logged in. There are many possible exploits. A critical exploit is to obtain root access by changing the user associated with the tty that the attacker is currently logged in as root, or by swapping ttys with an existing user logged in as root.

The concern is that many libraries and programs depend on the integrity of `/etc/utmp`. For example [Bis83], the `getlogin` routine in libc returns a pointer to the login name in `/etc/utmp`. However, to obtain that name, it first obtains the terminal name from `/etc/utmp` by determining if standard input is from a terminal, and if so, what the terminal's name is. If not, it searches for the terminals for standard output and standard error, in that order. However, the login name associated with the terminal is not necessarily the login name associated with the process if the `/etc/utmp` file has been altered. Because `/etc/utmp` was world-writable, the data was easy to alter.

Historically, since mail programs used `getlogin` for authentication, an attacker, logged in as user *sid*, could send *keith* a letter that appears to come from *matt*, currently logged in and using terminal 33. The attacker writes the message to a file named 'x' and issues the command:

```
sid% mail keith < x > /dev/tty33
```

The result is that the impersonation of *matt* is successful without any notification to *matt*, because mail gets the login name from standard output (terminal 33) and then prints no results to standard output, only standard error.

Given that many such exploits are possible, and that there are similar vulnerabilities to the one involving a world-writable `/etc/utmp`, this is a generalizable problem that can be modeled using our methods as follows. The first goal in the attack graph is altering the file:

$Grab\_TTY.requires$: $\{(local, local, ANY(euid : \mathbf{U}), \emptyset, Account : \mathbf{U}, tty = \mathbf{A} \wedge uid = \mathbf{U})\}$

$Grab\_TTY.provides$: $\{(local, ANY(IP), ANY(euid : \mathbf{U}), \emptyset, Account : \mathbf{U}, tty = \mathbf{A} \wedge uid = \mathbf{U2})\}$

The second goal is the use of the altered file:

$Use\_TTY.requires$: $\{(local, local, ANY, Read, Program : \mathbf{P}, Account : \mathbf{U}, tty = \mathbf{A} \wedge uid : \mathbf{U2})\}$

$Use\_TTY.provides$: $\{(local, ANY(IP), euid : \mathbf{U2}, ANY, Program : \mathbf{P}, ALL)\}$

Therefore, the result of applying the Analyze-Attack-Graph algorithm to both goals are as follows:

$\lambda(Grab\_TTY.requires, Grab\_TTY.provides)$:
$\{(Account : \mathbf{U}.(tty(\mathbf{A}) \wedge uid(\mathbf{U})), Account : \mathbf{U}.(tty(\mathbf{A}) \wedge uid(\mathbf{U2})), ANY(uid : \mathbf{U}), uid(\mathbf{U2}),$
$local, local, local, ANY(IP))\}$

The model indicates that the process to record *sufficient information* to analyze the attack is:

1. action(service.property): Localize the logging point at the `open` and `write` system calls in the kernel, to enable the ability to to monitor writes to the `utmp` file that result in changes to the associations between terminals and user IDs.

2. transition of credentials: From any uid:**U** to any other uid:**U2** (in the `utmp` file).

3. transition between src/dest network addresses: A connection to a terminal may be remote, for example, via an `ssh` connection. Therefore, the connection between the localhost and the remote machine needs to be monitored.

$\lambda(Use\_TTY.requires, Use\_TTY.provides)$:
$\{(Account : \mathbf{U}.(tty(\mathbf{A}) \wedge uid(\mathbf{U2})), ANY(Program : \mathbf{P}.ALL), ANY, euid : \mathbf{U2},$
$local, local, local, ANY(IP))\}$

The model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point at the `open` and `read` system calls in the kernel as well as additionally collecting the results of the `getlogin` system call in comparison to the uid used to `exec` the program. This enables recording the use of the modified user by a program.

2. transition of credentials: Comparison of the "real uid" to the uid:**U2** indicated in the `utmp` file.

3. transition between src/dest network addresses: A connection to a terminal may be remote, for example, via an `ssh` connection. Therefore, the connection between the localhost and the remote machine needs to be monitored.

To summarize:

Log the change in the ownership of a tty. This is done by recording writes to `/etc/utmp` and looking if one has changed ownership to root. Also record when other tty-related commands (syscalls or library calls) monitor/read the tty:uid relationship. One historical example is the `getlogin()` system call. The `wtmp` file would also be vulnerable, since it has both `lastlog` and `utmp` information. `wtmp` is used by `pam_lastlog`, `ac`, etc.....

Failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. For example, failing to record the writes to `utmp` causes the possible identity of the attacker to be missed. Failing to record commands that monitor the `utmp` file cause the forensic system to miss when the falsified information is read and potentially used. Therefore, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.7 Inconsistent Parameter Validation (e.g. with chsh or chfn)

Another early UNIX flaw occurred in `chsh` and `chfn`. Those programs would accept a new shell or full user name from the input line, even if the input contained a *colon* character.[6] The colon character is important in these files, because it is a field separator. Since the password would get written into the `/etc/passwd` file with the colons intact, a `uid` or `gid` of zero, indicating superuser privileges (or any arbitrary value), could be appended after a colon and overwrite the existing values. For example this line:

```
sean:x4nPA20z:500:500:Sean the Wizard:/home/sean:/bin/sh
```

could be changed to the following line:

```
sean:x4nPA20z:500:500:Sean the Wizard:/home/sean:/bin/sh:sean2::0:0...
```

(clipped due to page width) which is interpreted by the system as following *two* lines:

```
sean:x4nPA20z:500:500:Sean the Wizard:/home/sean:/bin/sh
sean2::0:0::/home/sean:/bin/sh
```

[6]See p. 47, slide #85 in *How Attackers Break Programs, and How to Write Programs More Securely* [Bis02] at `http://nob.cs.ucdavis.edu/bishop/secprog/sans2002.pdf`

The solution is either to have the program stop reading input from the user when it sees either a colon *or* a newline, or to have the program simply stop writing output to the `passwd` file when it sees a colon or newline.

A modern example of this flaw is the conflict between UNIX and Mac OS X (Darwin), wherein UNIX uses a forward slash ('/') to designate directories, whereas in Mac OS X, a file can be written with a forward slash contained in the name of a file or directory.

The capability pairs for this attack are to look for the user of these two programs (which can be invoked by any user):

Inconsistent_Param_Check.requires: $\{(\mathit{local}, \mathit{local}, \mathit{ANY}(\mathit{user}), \mathit{exec}, \texttt{chsh} \vee \texttt{chfn}, \mathit{code})\}$

This tuple indicates the need to record the content, looking at the specific fields and observing which are actually changed:

Inconsistent_Param_Check.provides:
$\{(\mathit{local}, \mathit{local}, \mathit{ANY}(\mathit{user}), \emptyset, \mathit{file}, \mathit{content}(\mathit{uid} : \mathbf{U} \vee \mathit{gid} : \mathbf{G} \vee \mathit{shell} : \mathbf{S}))\}$

The Analyze-Attack-Graph algorithm indicates that the process to record *sufficient information* to analyze the goal is:
$\lambda(\mathit{Inconsistent\_Param\_Check}) :$
$\{(\mathit{exec}(\texttt{chsh}.\mathit{code} \vee \texttt{chfn}.\mathit{code}), \mathit{file}(\mathit{content.uid} : \mathbf{U} \vee \mathit{content.gid} : \mathbf{G} \vee \mathit{content.shell} : \mathbf{S}),$
$\mathit{ANY}(\mathit{user}), \mathit{ANY}(\mathit{user}), \mathit{local}, \mathit{local}, \mathit{local}, \mathit{local},)\}$

1. action(service.property): Localize the logging point in the `exec` system call, such that the executions of the specific UNIX program(s) can be monitored and the current inputs given to those programs, in certain fields, can be compared with the input traditionally given. Such an analysis would indicate characters that are given as input that have never before been used as input for a given field, as well as other anomalous patterns.

2. transition of credentials: None. No credential change expected.

3. transition between src/dest network addresses: None. All addresses are local.

The result of the λ function is as follows: capture all calls and parameters and use data mining techniques to build a "normal" database of the parameters given to the `chsh` command. The results should include histograms of character frequencies given as input. We can assume that there will be large amounts of alphanumeric characters, and even many symbolic characters. Some characters (such as control characters) are disallowed and will not appear at all. If a colon character were to appear, even an automated detection system could know immediately that something unusual had been input and should be flagged for observation.

Failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. For example, failing to capture the `exec` calls removes the action with which to trigger the analysis of the inputs. Removing the recording of the inputs causes the information that could be compared with historical inputs to fail to be recorded, and thus analysis of the information is also prevented. Therefore, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.8 LAND Attack

The LAND attack[7] [Com98a, Fre98, m3l] involves spoofing TCP SYN packets [HB96] to have the source and destination addresses the same. Spoofed packets would escape certain filters and allow a DOS and/or crash of the system or routers because they would not be checked when the sender appeared to be localhost. This is an example of inconsistent parameter validation.

This attack has a single goal that we can easily monitor for, but requires two separate conditions be true. The first is simply the ability of the target machine to receive packets from any IP address, and and the second is the ability to spoof the source address in a packet sent to the target machine:

$LAND.requires$:
$\{(ANY(IP), local, \emptyset, receive, packet, TCP\_SYN),$
$(ANY(IP), local, \emptyset, spoof, packet, source = localhost)\}$

The capability provided is unusual in that it does not grant additional credentials, but causes a denial-of-service attack. However, the our language provides the ability to describe this:

$LAND.provides$: $\{(local, local, \emptyset, block, network\_stack, ALL)\}$

Applying the Analyze-Attack-Graph algorithm, this single-goal attack results in just one call to Analyze-Goal using the following output of λ. Since the requires set has two 6-tuples, the λ function results in 2 as well.

$\lambda(LAND.requires, LAND.provides)$:
$\{(receive(packet.TCP\_SYN), block(network\_stack.ALL), \emptyset, \emptyset, ANY(IP), local, local, local),$
$(spoof(packet.source = localhost), block(network\_stack.ALL), \emptyset, \emptyset,$
$ANY(IP), local, local, local)\}$

When we apply Analyze-Goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

[7] `http://www.insecure.org/sploits/land.ip.DOS.html`

1. action(service.property): Localize the logging point in the kernel, in the `receive` system call, to record the invocations of the call. The *block* action is different: it defines a process that is easy to characterize in broad terms, but difficult to define specifically, and harder to capture: how does one record denial of service on a machine that is being "denied" service? The reality is that this is an example of the flexibility of our model to specify this kind of action, but as the forensic analyst making the judgment of implementation, this is a goal that needs to be skipped on an ordinary FreeBSD machine without additional monitoring support beyond the scope of this dissertation. A third logging point is localized in the kernel socket calls to log the source and destination of all packets directed into the machine (though not "promiscuous" packets) and all packets directed from the machine to localhost. Then flag all packets that come in that do not have a correspond with a packet that was sent out. Since it is reasonable to expect that packets will have a short (in real-time perspective) lifespan, it is not necessary to keep packets sent out of the machine for longer than a few seconds, for comparative purposes.

2. transition of credentials: None. There is not actually a login.

3. transition between src/dest network addresses: (covered by the action).

Failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. For example, failing to record the `receive` call fails to record the actual action that is responsible for causing the exploit. Failing to record the IP addresses fails to provide the necessary data for the analysis to determine whether the IP addresses are the cause of the exploit. Failing to record the denial of service indicates that an analyst would not actually know if the events actually led to an exploit at all. Therefore, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit, though instructions to localize and record information that we have manually indicated are incomplete, and thus do not actually meet those necessary conditions. Therefore, while an implementation based on the model's output makes analyzing this attack theoretically possible, the lack of proper measurement tools on a UNIX system makes this attack difficult to analyze completely at present.

6.9 Shared Memory Code Injection

There are some other methods of attack that are more difficult to forensically analyze because they require resources not commonly available. Consider again the earlier example of executing a root shell. Another way of acquiring a root shell would be to inject code into the memory space of a process running as root. This could be done using a program not running as root if the non-root program and the program running as root were sharing memory together. In this case, although the shared memory allocation and memory locks are syscalls, the act of code injection is not. It is simply a memory write, which does not involve the kernel at all. No additional code would need to be executed, since the new commands which redirect control of the root shell, could be written directly to memory. Detecting such an attack requires monitoring all memory writes. Practically speaking, such a task probably requires specialized hardware support. The requires/provides model is still useful in modeling such a scenario, but is simply more difficult to implement practically. A capability pair for the attack looks like so:

Inject.requires:
$\{(local, local, euid = root, create, Process, shared\_memory : \mathbf{S}),$
$(local, local, ANY(user), write, Process, shared\_memory : \mathbf{S}),$
$(local, local, euid = root, read, Process, shared\_memory : \mathbf{S})\}$

Inject.provides: $\{local, local, ANY(user), modify, Process, running\_status\}$

Applying Analyze-Attack-Graph to the capability pair, the algorithm outputs the following:

$\lambda(Inject.requires, Inject.provides) :$
$\{$
$(create(Process.shared\_memory : \mathbf{S}), modify(Process.running\_status),$
$euid = root, ANY(user), local, local, local, local)$
$(write(Process.shared\_memory : \mathbf{S}), modify(Process.running\_status),$
$ANY(user), ANY(user), local, local, local, local)$
$(read(Process.shared\_memory : \mathbf{S}), modify(Process.running\_status),$
$euid = root, ANY(user), local, local, local, local)$
$\}$

The model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the `shm_open` system call to determine where the shared memory object is created and under what mode and permissions. Monitor (probably in hardware or a virtual machine) memory reads and writes. Monitor changes in the operation of the root-owned program based on the TOCTTOU exploit of the shared memory, perhaps by using anomaly detection techniques to look for statistical variations of the operating conditions of the software. Otherwise, the exploit must be treated as a race condition and analyzed based on the *requires* capability set, rather than the *provides* set.

2. transition of credentials: A user with an euid of root interacting with the shared memory space with another user.

3. transition between src/dest network addresses: None. All addresses are local.

Failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. For example, failing to record the `shm_open` system call and the start and size of the memory space created fails to indicate the action that begins the series of events, and the memory region to monitor with regard to future events. Failing to record memory reads and writes, the memory addresses, and the users that do the reading and writing means the changes in the information that could lead to the race condition are missed. Therefore, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.10 The 1988 Internet Worm

The 1988 Internet Worm [Bis88b, ER89, See89a, See89b, Spa89], a classic multi-stage, multi-exploit attack, exploited a buffer overflow vulnerability in `fingerd`, and several problems with other UNIX programs to break into systems. The attack caused denials of service by propagating to as many machines as possible, causing the systems to be swamped and unusable. We use our model to show what additional data would have simplified analysis of the worm.

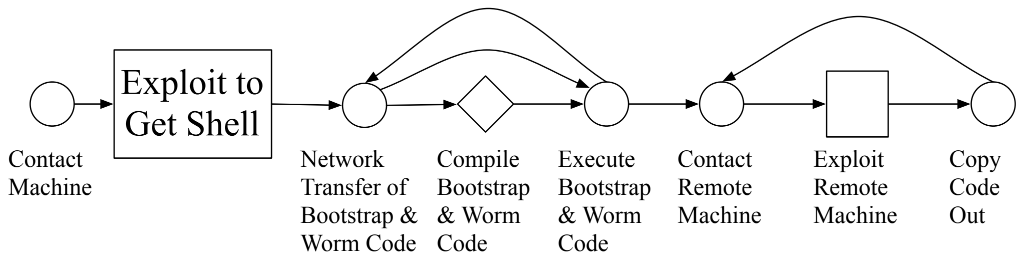

Exploit to Get Shell
Contact Machine
Network Transfer of Bootstrap & Worm Code
Compile Bootstrap & Worm Code
Execute Bootstrap & Worm Code
Contact Remote Machine
Exploit Remote Machine
Copy Code Out

Figure 6.4: The attack graph used by the Internet Worm

For the requires portion of the initial contact, intuitively, contacting a machine is a simple process. It requires no local permissions, and merely a process running on the target machine that is listening on an open network port. For the provides portion, the capabilities resulting from contacting a machine are limited, and unclear until an exploit occurs. Therefore the capability pairs, as well as the outputs of the ANALYZE-ATTACK-GRAPH algorithm, result in the following:

ContactIn.requires: $\{(ANY(IP), local, \emptyset, Socket.listen, Net\_Process : \mathbf{P}, code)\}$
where $Net\_Process$ is a process on the target machine that has a program listening on an open network port.

ContactIn.provides: $\{(ANY(IP), local, \emptyset, Communicate.recv, Net\_Process : \mathbf{P}, code)\}$
$\lambda(ContactIn.requies, ContactIn.provides)$:
$\{(Communicate.listen(Net\_Process : \mathbf{P}.code), Communicate.recv(Net\_Process : \mathbf{P}.code), \emptyset, \emptyset,$
$ANY(IP), ANY(IP), local, local)\}$
where $Net(Process)$ is a function that describes a process that has an open network port.

The initial contact, intuitively, requires no local permissions, and merely a process running on the target machine listening on an open network port. The capabilities resulting from contacting a machine are limited, and unclear until an exploit occurs. Therefore the Analyze-Goal algorithm indicates that the process to record information *sufficient* to analyze the goal, based on the capability pair, is:

1. action(service.property): Localize the logging point in the `accept` system call in the kernel. Look for a program that is listening for a socket connection and then begins receiving communication from a remote machine.

2. transition of credentials: $\emptyset$ and $\emptyset$ indicate that no local credentials are required for either capability

3. transition between src/dest network addresses: The goal indicates communication from a remote source machine, so its address must be captured.

The *TransferIn* goal is slightly more complicated. To transfer a readable file to the target machine, the attacker needs to be a user on the local machine and on the target machine, and have a process capable of receiving network data from a remote machine on the target machine (e.g. ftp). After performing the transfer, the user has a readable file on the target machine:

TransferIn.requires: $\{(ANY(IP), local, uid : \mathbf{u}, Communicate.recv, file(ANY), code)\}$

TransferIn.provides: $\{(local, local, uid : \mathbf{u}, read \vee (read \wedge exec), file(\mathbf{Worm}), code)\}$
$\lambda(TransferIn.requies, TransferIn.provides)$:
$\{(Communicate.recv(file(ANY).code), read(file(ANY).code), uid : \mathbf{u}, uid : \mathbf{u},$
$ANY(IP), local, local, local)\}$

Applying Analyze-Goal to *TransferIn*, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the `read` system call in the kernel.
2. transition of credentials: Record the user ID **u** that owns the program previously exploited.
3. transition between src/dest network addresses: Record the address of the machine from which the data is being transferred.

Though we cannot precisely define the exploit to get a shell, we can determine its bounds by applying the τ function to the *provides* capabilities of the *ContactIn* goal and the *requires* capabilities of the *TransferIn* goal:

$$\tau(ContactIn.Provides, TransferIn.requires)$$

$$= (ExploitLocal_{max}.requires, ExploitLocal_{min}.provides)$$

$\lambda(ExploitLocal_{max}.requires, ExploitLocal_{min}.provides)$:
$\{(Communicate.recv(Net\_Process : \mathbf{P}), Communicate.recv(file(ANY).code), uid : \mathbf{u}, uid : \mathbf{u},$
$ANY(IP), local, local, local)\}$

So, somewhere between the two events, the attacker acquires the ability to transfer a file, readable by a particular local user. Therefore, the bounds on the exploit involve a specific set of IP addresses that are used to exploit the target machine and acquire a command interpreter of some sort, and provide the ability to execute programs capable of requesting or receiving a file.

Applying Analyze-Goal to *ExploitLocal*, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Not easily boundable by the action. See transition of credentials.
2. transition of credentials: Record the user ID **u** of the program previously that received the socket connection, the program used to transfer data onto the target system, and the uid of any programs in between.
3. transition between src/dest network addresses: None; all addresses are local.

The *Compile* goal is optional. That is, it may be trivially achieved if the program transfered to the target machine is a script that does not require compilation, and thus can simply be executed, as usual. The goal ensures that a program can be executed. The requires portion must indicate that a local user can read the worm code, execute the compiler, and write a binary. The provides portion indicates that after compiling, a local user can execute the binary.

Compile.requires:
$\{(local, local, uid : \mathbf{u}, exec, file(\mathbf{Compiler}), code),$
$(local, local, uid : \mathbf{u}, read, file(\mathbf{Worm}), code)\}$

Compile.provides: $\{(local, local, uid : \mathbf{u}, exec, file(\mathbf{Worm}), code)\}$
$\lambda(Compile.requies, Compile.provides) :$
$\{(exec(file(\mathbf{Compiler}).code), exec(file(\mathbf{Worm}).code), uid : \mathbf{u}, uid : \mathbf{u}, local, local, local, local)$
$(read(file(\mathbf{Worm}).code), exec(file(\mathbf{Worm}).code), uid : \mathbf{u}, uid : \mathbf{u}, local, local, local, local)\}$

Applying Analyze-Goal to the *Compile* goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): The logging point is in both the `stat` and `exec` system calls to check to see if the file transferred previously has read/exec permissions for uid:**u**, and also whether a compiler has been run.

2. transition of credentials: The uid/gid of the transferred file and the euid/egid of the compiler being run (if any).

3. transition between src/dest network addresses: All addresses are local. Nothing to record.

The intuition for the *Execute* goal, that starts the program, is as follows: the requires portion indicates that a local user can execute the binary, and the provides portion indicates a running binary (the worm code) owned by a local user.

Execute.requires: $\{(local, local, uid : \mathbf{u}, exec, file(\mathbf{Worm}), code)\}$
Execute.provides: $\{(local, local, uid : \mathbf{u}, run, file(\mathbf{Worm}), code)\}$
$\lambda(Execute.requies, Execute.provides) :$
$\{(exec(file(\mathbf{Worm}).code), run(file(\mathbf{Worm}).code), uid : \mathbf{u}, uid : \mathbf{u}, local, local, local, local)\}$

Applying Analyze-Goal to the *Execute* goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging points in the kernel in the `exec` and `write` system calls.

2. transition of credentials: None. Record the user ID **u** of the program being executed.

3. transition between src/dest network addresses: All addresses are local. Nothing to record.

The *ContactOut* goal starts the propagation of the worm. Intuitively, this requires a local user who owns a running program, and the running program provides the ability to communicate with other machines on the network.

ContactOut.requires: $\{(local, ANY(IP), uid : \mathbf{u}, Run, Program : \mathbf{Worm}), code)\}$

ContactOut.provides: $\{(local, ANY(IP), uid : \mathbf{u}, Communicate.send, ANY, ANY)\}$
$\lambda(ContactOut.requies, ContactOut.provides)$:
$\{(Exec(Program : \mathbf{Worm}.code), Communicate.send(ANY.ANY), uid : \mathbf{u},$
$local, ANY(IP), local, ANY(IP))\}$

Applying Analyze-Goal to the ContactOut goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the `connect` system call in the kernel.

2. transition of credentials: The user ID **u** of the program initiating the socket connection with the remote machine.

3. transition between src/dest network addresses: The network address of the machine being contacted.

The *TransferOut* goal involves transferring the worm code to the remote target. The capability required is a shell owned by a local user on the remote target. The capability provided on the *local* machine is nothing, since the code was simply transferred. That code provides a capability on the *remote* machine, but we do not model the remote target, just the local target.

TransferOut.requires: $\{(local, ANY(IP), \emptyset, Communicate.send, file(\mathbf{Worm}), code)\}$

TransferOut.provides: $\{(local, ANY(IP), \emptyset, \emptyset, \emptyset, \emptyset)\}$
$\lambda(TransferOut.requies, TransferOut.provides)$:
$\{(Communicate.send(file(\mathbf{Worm}).code), \emptyset, \emptyset, \emptyset, local, ANY(IP), local, ANY(IP))\}$

Applying Analyze-Goal to the *TransferOut* goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the `send` system call, triggered when data is sent to a remote network address.

2. transition of credentials: The user ID **u** of the program being used to transfer data.

3. transition between src/dest network addresses: The network address of the machine having data transferred to it.

The *ExploitRemote* goal is, again, unknown, but can be estimated by using bounds from the previous and succeeding goals:

$$\tau(\mathit{ContactOut}.Provides, TransferOut.requires)$$

$$= (\mathit{ExploitRemote}_{max}.requires, \mathit{ExploitRemote}_{min}.provides)$$

$\lambda(\mathit{ExploitRemote}_{max}.requires, \mathit{ExploitRemote}_{min}.provides)$:
$\{(Communicate.send(ANY.ANY), Communicate.send(file(\mathbf{Worm}).code),$
$local, ANY(IP), local, ANY(IP))\}$

That is, the requires portion of *ExploitRemote* is the ability to communicate with other machines on the network, and what is provided is a shell owned by a local user on the remote target.

Applying Analyze-Goal to the *ExploitRemote* goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the kernel socket calls, to monitor socket openings and connections to send the worm code to remote machines.

2. transition of credentials: The provides portion of the goal adds no credentials, but the euid of the other programs run by the user making the outgoing network connections still need to be confirmed.

3. transition between src/dest network addresses: The network address of the remote machine being exploited.

The common denominator of the information that must be extracted is actually a small subset of system calls with a small amount of contextual information relating to user IDs, file paths, and network addresses. This confirms that in many cases, only a small, very focused amount of information is actually necessary. Nonetheless, this information is infrequently captured in existing systems.

In summary, the calls that must be recorded are the `accept`, `connect`, and `send` socket calls (and related IP addresses); the `read` system call (and related users, programs, and IP addresses), and the `stat`, `exec`, and `write` system calls (and related programs, users, and files opened). Failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. For example, failing to record the `accept` call (and its result, indicating success or failure) would cause the system to miss the connection of another machine with the localhost. Failing to record the `read` system call causes the system to miss the system receiving the worm code from the attacking machine. Failing to record the `stat`, `exec`, and `write` system calls cause the system to miss the series of events that show a chain from the transfer of the code to the machine to its execution. While each alone do not show a great deal of value themselves, the combination to form an unbroken chain of events is key, Failing to record the `send` call causes the system to miss the propagation of the worm, by sending the worm code to other hosts. Therefore, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.10.1 Testing τ with λ

The knowledge of several events in a row presents an opportunity to mathematically test the validity of the τ function by comparing the output of $\lambda(\tau(TransferIn.provides, Execute.requires))$, which should be $\lambda(Compile_{max}, Compile_{min})$, with the output of $\lambda(Compile.requires, Compile.provides)$.

Let $x = TransferIn$, $z = Execute$, and $y = Compile$. We see:

$$\lambda(\tau(x.provides, z.requires)) = \lambda(y_{max}.requires, y_{min}.provides)$$
$$= \{(read(file(\mathbf{worm}).code) \vee (read \wedge exec)(file(\mathbf{worm}).code)), exec(file(\mathbf{worm}).code),$$
$$uid : \mathbf{u}, uid : \mathbf{u}, local, local, local, local.)\}$$

The result of comparing the output of the two λ functions and removing the overlap indicates that actual and generated forms of y are nearly identical:

$$\lambda(y.requires, y.provides) :$$
$$\{(read(file(\mathbf{worm}).code) \vee exec(file(\mathbf{Compiler}).code)), exec(file(\mathbf{worm}).code),$$
$$uid : \mathbf{u}, uid : \mathbf{u}, local, local, local, local)\}$$

The difference between $\lambda(y_{max}, y_{min})$ and $\lambda(y.requires, y.provides)$ is as the model predicts — indeed, $y.requires \subseteq y_{max}.requires$ and $y_{min}.provides = y.provides$.

6.11 Christma Exec Worm

The "Chrisma Exec worm" that appeared in 1987,[8] and infected IBM VM/CMS systems, contained elements of both worms and Trojan horses. A program, which, like a shell script, did not require compilation to execute, was sent via email. When a user retrieved and executed the file, it would open and read two files on the system ("NETLOG" and "NAMES"), looking for email addresses (some of which may be mailing lists), and then would send itself to all email addresses it finds. Presumably all the email addresses would also be valid. The program would probably not have been damaging other than for the way in which the computing and network infrastructure, namely BITNET, was set up at the time ("store-and-forward").

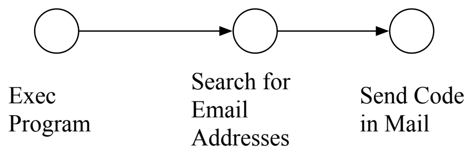


Figure 6.5: The attack graph used by the Christma Exec Worm

Another "worm" was subsequently sent that actually attempted to ameliorate the spread of the Christma Exec virus by erradicating it on users' systems.

The sub-goals can be modeled as follows:

$Execute.requires$: $\{(local, local, ANY(uid), exec, program : \mathbf{P}, code)\}$

$Execute.provides$: $\{(local, local, ANY(uid), exec, program : \mathbf{P}, code)$,
$(local, local, uid : \mathbf{u}, read, files(\ldots), content)\}$

$\lambda(Execute.requies, Execute.provides) :$
$\{(exec(program : \mathbf{P}.code), exec(program : \mathbf{P}.code), ANY(uid), ANY(uid),$
$local, local, local, local),$
$exec(program : \mathbf{P}.code), read(files(\ldots).content), uid : \mathbf{u}, ANY(uid),$
$local, local, local, local)\}$

The model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the kernel in the `exec` system calls to observe the execution of the program (Trojan horse) by the local user.

[8]See Issues 5.72, 5.74, 5.79, 5.80, 5.81, and 6.1—6.7 of the RISKS Digest [Neu] and also digest issues 5-176 through 5-206 of the Virus-L mailing list (the digest version of the `comp.virus` newsgroup).

2. transition of credentials: The uid:**u** of the user during execution.

3. transition between src/dest network addresses: None. All addresses are local.

Search.requires: $\{(local, local, uid : \mathbf{u}, read, files(\ldots), content)\}$

Search.provides: $\{(local, local, \mathbf{u}, know, mail, addresses)\}$

$\lambda(Search.requies, Search.provides)$:
$\{(read(files(\ldots).content), know(mail.addresses), uid : \mathbf{u}, \mathbf{u},$
$local, local, local, local)\}$

The model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the kernel in the `open` and `read` system calls that are made by the Trojan program to determine what files it is searching (for usernames or email addresses, for example). The specification of $know(mail.addresses)$ does not lead to anything that can be logged immediately. These addresses are held for the next goal.

2. transition of credentials: None. No change in uid.

3. transition between src/dest network addresses: All addresses are local.

Send.requires: $\{(local, local, uid : \mathbf{u}, exec, SENDFILE, code),$
$(local, local, uid : \mathbf{u}, send, mail, addresses)\}$

Send.provides: $\{(local, ANY(IP), \emptyset, \emptyset, script : \mathbf{s}, code)\}$

$\lambda(Send.requies, Send.provides)$:
$\{(exec(SENDFILE.code), \emptyset(script : \mathbf{s}.code), uid : \mathbf{u}, \emptyset, local, local, local, ANY(IP)),$
$(send(mail.addresses), \emptyset(script : \mathbf{s}.code), uid : \mathbf{u}, \emptyset, local, local, local, ANY(IP))\}$

The model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the `exec` system call in the kernel to capture executions of `SENDFILE`.[9] Also, the previous goal indicates the logging point

[9]BITNET and VNET, the networks used by the IBM VM/CMS systems, are store-and-forward networks, which make sending mail the same process as file transfer. Hence, the virus was propagated by a file transfer program, SENDFILE, and no specific mail program was needed.

as a set of function calls within the SENDFILE program, to capture for mail sent to the addresses collected in the previous goal. The $\emptyset(script : \mathbf{s}.code)$ specification does not provide an action and therefore does not provide a condition that can be monitored.

2. transition of credentials: None. No change in uid.

3. transition between src/dest network addresses: The IP addresses of the machines that the worm contacts, in order to spread.

In summary, the `exec`, `open`, and `read` system calls must be captured. Failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. For example, failing to record the `exec` system call causes the system to miss the act of executing the malicious program. Failing to record the `open` call causes the system to miss the filehandle used by the `read` system call, which must be recorded or the key act of the malicious program scanning "address book" files on the system is missed. Therefore, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.12 NFS Exploits

A number of well-known exploits [Com94a, Com98b, van94, van98] have been used against the Network File System (NFS) [Sun89], as a result of implementation bugs and configuration errors. Indeed, at the San Diego Supercomputer Center (SDSC) in 2004 [MB05, Sin05], an attack occurred in which the key flaw exploited was that a configuration setting enabled NFS volumes to be mounted on "high ports," which are not restricted only to the superuser. Given this, and the fact that NFS-shared volumes can only limit the ability to mount a filesystem by IP address, and not by user, the filesystem could be mounted and manipulated by any user. Even if the NFS daemon had not allowed mounting on "high ports," the filesystem could still have been mounted on "low" ports by a user logged in as `root` on an untrusted system that was nevertheless in the range of IP addresses allowed by the NFS server. Indeed, there are reasons why such restrictions may not be desirable.

By virtue of the way that NFS is designed, it is not possible to avoid all illicit access. It was simply not designed with high security in mind. However, the accesses and other operations can be logged if our forensic model is applied to attack graphs based on violating policies that NFS is supposed to follow. At SDSC, a minimum of information was recorded using standard forensic means: merely the IP address of the machine requesting an NFS mount and the time at which it occurred. But this is not enough information to analyze the attack.

An analysis of the logging and auditing mechanisms necessary to detect the policy violations that result from the exploit of some of the vulnerabilities in NFS v2 has been previously done by Bishop, et al.[10] The result of the analysis was a list of policies the exploits would violate and the resulting data that would need to be recorded and analyzed in order to determine how the exploit was actually carried out. The Logging and Auditing File System (LAFS) [Wee95] is an implementation of a logging system built on top of NFS. It uses its own policy language to allow a system administrator to define the policy violations to log. In doing so, it attempts to ameliorate the absence of security logging built into NFS.

The basic ideas behind the historical NFS exploits generally either involved illicitly obtaining a filehandle, or legitimately obtaining a filehandle and then illicitly accessing files. An attack graph demonstrating these exploit approaches appears in Figure 6.6.

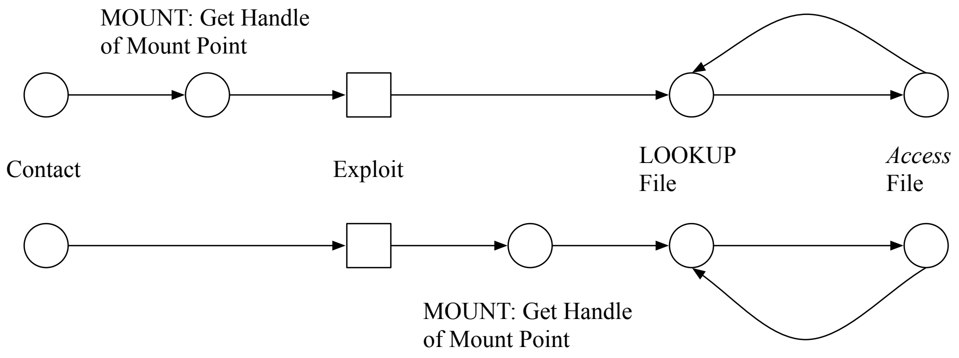

MOUNT: Get Handle of Mount Point
Contact
Exploit
LOOKUP File
Access File
MOUNT: Get Handle of Mount Point

Figure 6.6: Attack graphs for two possible classes of NFS exploits.

Within the NFS framework, there are several operations that return a filehandle: CREATE, MKDIR, and LOOKUP. There are several more operations that return a nonfile handle: GETATTR, SETATTR, READ, WRITE, REMOVE, RENAME, LINK, SYMLINK, READLINK, RMDIR, READDR, and STATFS.

By using our own forensic model with attack graphs based on violations of standard security policies, our methods would capture identical data to that in the work by Bishop, et al. The difference lies in how the model could be derived (directly, given policy translation) and how the directives would be represented (as requires/provides) capability pairs. The capability pairs for each goal in the attack graph appear as follows:

$\mathit{Contact.requires}$: $\{(\mathit{ANY}(\mathit{IP}) : \mathbf{A}, \mathit{local}, \emptyset, \mathit{Listen}, \texttt{nfsd}, \mathit{code})\}$

[10]See §24.6.1, "Audit Analysis of the NFS Version 2 Protocol," in *Computer Security: Art and Science* [Bis03].

$Contact.provides$: $\{(ANY(IP):\mathbf{A}, local, \emptyset, Request, \texttt{nfsd}, code)\}$

$\lambda(Contact.requies, Contact.provides)$:
$\{(Listen(\texttt{nfsd}.code), Request(\texttt{nfsd}.code), \emptyset, \emptyset, ANY(IP):\mathbf{A}, ANY(IP):\mathbf{A}, local, local)\}$

Applying ANALYZE-GOAL to the *Contact* goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the kernel's socket-related `listen` system call and the `mount` call, indicating the request issued to NFS and the local filesystem.
2. transition of credentials: None. No permissions changes or violations in this goal.
3. transition between src/dest network addresses: The incoming connection, ANY(IP):**A** to the localhost.

We apply the τ function to the *ExploitBefore* goal based on *Contact.provides* and *Mount.requres*:

$$\tau(Contact.provides, Mount.requires)$$
$$= (ExploitBefore.requires_{max}, ExploitBefore.provides_{min})$$

To define this, we first analyze *Mount* and then come back to *ExploitBefore*.
$Mount.requires$:
$\{((ANY(\mathbf{ACL}):\mathbf{L} \wedge ANY(\mathbf{PortList}):\mathbf{P}, local, ANY(UID), running, \texttt{nfsd}, \emptyset)\}$
where $\mathbf{L} \in \mathbf{ACL} \subseteq ANY(IP)$ and is specified in the `access.deny` list; and where $\mathbf{P} \in \mathbf{PortList}$, and $\mathbf{PortList} = \mathbb{N} \vee \mathbb{N} < 1024$, depending on configuration settings of `nfsd` on the target machine.

$Mount.provides$: $\{(local, local, uid:\mathbf{u1}, Access, NFS\_Mounted\_Filesystem:\mathbf{n1}, ALL)\}$

$\lambda(Mount.requies, Mount.provides)$:
$\{(running(\texttt{nfsd}.\emptyset), Access(NFS\_Mounted\_Filesystem:\mathbf{n1}.ALL), ANY(UID), uid:\mathbf{u1},$
$ANY(\mathbf{ACL}):\mathbf{L}, local, local, local)\}$

Therefore, applying ANALYZE-GOAL to the *Mount* goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the `nfsd` code to monitor for `Access` calls to the `n1` filesystem, while `nfsd` is running.

2. transition of credentials: None; but we still must monitor the $uid : \mathbf{u1}$ that the "attacking" machine presents to `nfsd`.

3. transition between src/dest network addresses: From the machine mounting the `nfs`-shared directory to the local machine, as well as the port number of the machine requesting the mount, as the port number can indicate whether a connection is requested from a privileged user.

We also must apply Analyze-Goal to the *ExploitBefore* goal, now that it has been bounded:

$\lambda(ExploitBefore.requires_{max}, ExploitBefore.provides_{min})$:
$\{(Request(\texttt{nfsd}.code), running(\texttt{nfsd}.\emptyset), \emptyset, ANY(UID), ANY(IP) : \mathbf{A},$
$local, ANY(\mathbf{ACL}) : \mathbf{L} \wedge ANY(\mathbf{PortList}) : \mathbf{P}, local)\}$

The model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the `mount` and `exec` system calls to monitor whether a mount request occurs and whether the result causes `nfsd` to execute.

2. transition of credentials: From no local login to any local user.

3. transition between src/dest network addresses: Between ANY(IP):**A**, ANY(**ACL**):**L** (including port number), and localhost.

We apply the τ function to the *ExploitAfter* goal based *Mount.provides* and *Lookup.requires*:

$$\tau(Mount.provides, Lookup.requires)$$

$$= (ExploitAfter.requires_{max}, ExploitAfter.provides_{min})$$

We first analyze *Lookup*:

Lookup.requires: $\{(ANY(IP), local, uid : \mathbf{u2}, Access, NFS\_Mounted\_Filesystem : \mathbf{n2}, ALL)\}$

Lookup.provides: $\{(ANY(IP), local, uid : \mathbf{u2}, Access, FileHandle : \mathbf{n3}, ALL)\}$
where **n3** is a filehandle of a file on the NFS-mounted system whose filehandle is mounted as **n2**.

$\lambda(Lookup.requires, Lookup.provides)$:
$\{(Access(NFS\_Mounted\_Filesystem : \mathbf{n2}.ALL), Access(FileHandle : \mathbf{n3}.ALL),$
$uid : \mathbf{u2}, uid : \mathbf{u2}, ANY(IP), ANY(IP), local, local)\}$

We also must apply Analyze-Goal to the *ExploitAfter* goal, now that it, too, has been bounded:

$\lambda(ExploitAfter.requires_{max}, ExploitAfter.provides_{min})$:
$\{(Access(NFS\_Mounted\_Filesystem : \mathbf{n1}.ALL), Access(NFS\_Mounted\_Filesystem : \mathbf{n2}.ALL),$
$uid : \mathbf{u1}, uid : \mathbf{u2}, local, ANY(IP), local, local)\}$

The model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Both actions are *Access*, but to potentially different objects. The primary transition is in the credentials. Therefore, localize the logging point in the access call in `nfsd` to capture a change in credentials.

2. transition of credentials: From uid:**u1** to uid:**u2**.

3. transition between src/dest network addresses: None; all addresses are local.

Therefore, applying Analyze-Goal to the *Lookup* goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the kernel to capture the filesystem, files, and credentials when a user accesses FileHandle:**n3** on Filesystem:**n2**.

2. transition of credentials: None — all uid:**u2**.

3. transition between src/dest network addresses: Between ANY(IP) and the localhost.

Finally, we look at the *Access* goal:
Access.requires: $\{(ANY(IP), local, uid : \mathbf{u2}, Access, FileHandle : \mathbf{n3}, ALL)\}$

Access.provides: $\{(ANY(IP), local, \emptyset, Know \wedge Modify, File : \mathbf{n3}, ALL)\}$

where *File:***n3** is the file whose filehandle is **n3**.

$\lambda(Access.requies, Access.provides)$:
$\{(Access(FileHandle : \mathbf{n3}.ALL), (Know \wedge Modify)(File : \mathbf{n3}.ALL), uid : \mathbf{u2}, \emptyset,$
$ANY(IP), ANY(IP), local, local)\}$

Therefore, applying Analyze-Goal to the *Access* goal, the model indicates that the process to record *sufficient information* to analyze the goal is:

1. action(service.property): Localize the logging point in the kernel to record the `write` system call (and other calls related to changing files permissions or locations), showing when a user with FileHandle:**n3** modifies the associated file.

2. transition of credentials: None — access is with uid:**u2**.

3. transition between src/dest network addresses: From ANY(IP) to localhost.

Note that if $\mathbf{u2} \subseteq \mathbf{u1}$ and $\mathbf{n2} \subseteq \mathbf{n1}$, *ExploitAfter* is trivially met. Similarly, if $\mathbf{A} \in \mathbf{ACL} \wedge \mathbf{P} \in \mathbf{PortList}$, *ExploitBefore* is trivially met. Trivially meeting either goal pre-empts the need for use of an actual exploit and instead may represent a configuration error or legitimate user.

Implementing a logging system to analyze NFS can be more complicated than analyzing many other UNIX systems because NFS is reliant not only on UNIX permissions and a site security policy but on its own configuration file (`/etc/exports` on FreeBSD).

In summary, the `listen`, `mount`, `exec`, and `write` library calls; and the `Access` call in `nfsd`, as well as related IP addresses, users, and file permissions, are necessary to analyze the attack. Failing to record the information indicated by the model results in an insufficient amount of data to analyze the attack. For example, failing to record the transition from the `listen` call to the `mount` call fail to record one of the two possible key exploits that may occur on the system to illicitly gain access to the filesystem. Failing to record the `Access`, `exec`, and `write` calls would cause the system to miss the other possible exploit, resulting in access not just to the filesystem, but the files on it, and the recording of the results of that breach in security. Therefore, the set of information specified by the model represents a *necessary condition* to be able to analyze this exploit.

6.13 Summary of Examples

This chapter presented twelve examples of applying a forensic model, Laocoön, with a set of qualities that enhance the forensic analyzability of the system and are derived from a number of principles of forensic analysis. In each case, we have indicated why the forensic information indicated by the model represents a *necessary condition* to being able to analyze the attack. In the next chapter, we show a validation of the model by way of the results of implementing several of these examples, and discuss why the data recorded represents a *sufficient condition* for analyzing each attack.

7

Implementation, Experiments, and Results

Sherlock Holmes: *"Data! Data! Data! ... I can't make bricks without clay."*
—Sir Arthur Conan Doyle, "The Adventure of the Copper Beeches,"
The Strand Magazine (1892)

In this chapter, as a validation of our forensic model, we show the results of implementing several of the examples discussed in previous chapter. We implemented the logging mechanisms to capture the data prescribed by the model on a FreeBSD 5.x system, in a variety of ways. Some information could be gathered with simple BSM logs, with `ktrace/strace` (system calls) or `ltrace` (dynamic library calls). In our experiments, we instrumented the kernel to record the calls and provide additional information such as the contextual information tied to a network connection being received.

In the previous chapter, we showed a number of multi-step attack graphs. In the results for the implementations of those attacks here, we correlate events in each attack and assign a path identifier (which we do not show). This is mentioned in each multi-step example.

7.1 Obtaining a Local Root Shell

As we discussed earlier, there are many possible ways to get root privileges, and among them both local and remote exploits. We examine a local exploit resulting in a root shell, which is the second half of the example shown in Section 6.1. Consider the following piece of code [fid01]:

```
int main(int argc, char *argv[]) {
        char buffer[500];
        strcpy(buffer, argv[1]);
        printf("Safe program?");
        return 0;
}
```

We ran this program, called `vulnerable`, in two different ways. The first way is "safe" and does not exploit the vulnerability. It simply passes 499 instances of the ASCII character 'a' as the argument to `vulnerable`. The second way exploits the vulnerability by copying more than 500 characters into the buffer. The characters include *shellcode* that returns a shell owned by `root`.

The results of applying the ANALYZE-ATTACK-GRAPH algorithm to the *Exploit*[1] and *Root_Shell_Local* capability pairs in the attack graph in Section 6.1 tells us that the the forensic system must simply monitor program executions and their uid, effective uid (euid), and group identification (gid).

The following table gives this information when the vulnerability is exploited. It shows the results of implementing the model for this exploit, and collecting the data specified when the vulnerability is exploited. `exploit` refers to the "wrapper program" that exploits `vulnerable`:

Prog	PID	R/EUID	RGID	Syscall
`exploit`	20352	1001/1001	1001	execve
`vulnerable`	20352	1001/0	1001	execve
`sh`	20352	1001/0	1001	execve
`rm`	20353	1001/0	1001	execve

The table shows that `exploit` leads directly to a change in privileges when it `exec`'s the program `vulnerable`. Likewise, we see that `sh` must be `exec`'d by `vulnerable` because the PID stays the same, as do the permissions and the address (localhost).[2] The syscalls continue with the elevated permissions and the use of the `rm` program, showing therein what damage an intruder may be doing.

For comparison, we executed the "safe" version of this program from a wrapper program called `normal`. In this case, the elevated privileges end after the `vulnerable` program ends and the PID counter increments (for the `execve` immediately after the `vulnerable` program), suggesting no successful `exec`, spawned from `vulnerable`, occurred:

Prog	PID	R/EUID	RGID	Syscall
normal	20380	1001/1001	1001	execve
vulnerable	20380	1001/0	1001	execve

[1]Previously bounded using the BOUND-UNKNOWNS algorithm.
[2]Together, these plus a unique number, form the unique path identifier (UPI).

These results show the means of attaining superuser permissions as well as the actions that take place as a result of attaining them. Additionally, the results show that they do so with a relatively small amount of data. More data is unnecessary, and less data would have resulted in the loss of useful information. Though the results do not show the exact nature of the exploit, this is by design: doing so would have required very low-level, and high-cost information to gather. Therefore, the results show where to look to find the flaw, rather than the flaw itself. Most importantly, the results allow the exploit to be analyzed and they also match the data indicated by the model. Therefore, the results of this experiment validate the model for this exploit.

Alternatively, consider a variant of this attack that involves the following series of events:

1. Compromise the privileged process

2. Copy the shell with the appropriate permissions (setuid root) where it is executable by a normal user.

3. Fork the compromised process.

4. Drop privileges in the child process.

5. Invoke the setuid root shell.

In this case, by ensuring that *all* process invocations are captured — that is, `execve` *and fork*, the results would be similar to the ones shown above and would show the exec of the exploit, the exec of the vulnerable program, the exec of the copy command, the fork, of the compromised process, and the invocation of the shell, linked by the unique path identifier.

7.2 Spyware via a Trojaned sshd

To detect spyware capturing and re-sending a password, the results of applying the Analyze-Attack-Graph algorithm to the *Capture_Password*, *Intermediate*,[3] and *Send_Password* goals, defined in Section 6.2 is a directive to first record the capture, and then the send. Capturing requires the user to enter their password into a program that calls `pam_authenticate`. The sending is opening and using a socket to send the password.

Prog	Call	Arg2	Arg4.sa_data	Return Value
`ssh`	`pam_authenticate`			0
`ssh`	`socket`			0
`ssh`	`sendto`	MyPasswd	`192.168.0.1`	8

[3]Previously bounded using the Bound-Unknowns algorithm.

In the data above,[4] we see exactly the routine used to capture the password (line 1), the routine to open the socket (line 2) , the sending of the password, the destination of the spyware results, and the return value, 8, indicating the number of characters sent (all on line 3). In this case, the program capturing the password did not communicate with another program to send the password; the capturing program also did the sending itself.

These results show the means of capturing and sending a password surreptitiously, as well as the events that may occur in between the two events to prevent an analyst from noticing a correlation between the two events. Again, the results show that this is possible with relatively small amounts of data. The results also match the data indicated by the model. Therefore, the results of this experiment validate the model for this exploit.

7.3 Modify /etc/passwd via lpr bug

The application of Analyze-Attack-Graph to the attack graph described in Section 6.3, indicated a required set of data.[5] Instrumenting the system accordingly, and running the exploit results in the following results. Notice the `open()` system call when `lpr` reads (and accepts) a temporary file that we have written arbitrarily:

Prog	R/EUID/RGID	Syscall	Arg1 (file)	Arg2 (flags)	FileUID
lpr	1001/10011001	open	/tmp/.tmp.477	READ	1001

We really want to see the symbolic link from the spool file (which is owned by root) to the temporary file (also owned by root). Given that the link created is owned by root, we also obviously see this process is run as root:

Prog	R/EUID RGID	Syscall	Arg1 (link)	Arg2 (target)	File1UID	File2UID
lpr	1001/0 1001	symlink	/tmp/.tmp.477	/var/spool/dfA292	0	0

We then see that file removed (after lpr has accepted the link):

Prog	R/EUID/RGID	Syscall	Arg1 (file)	FileU
rm	1001/1001/1001	unlink	/tmp/.tmp.477	1001

And we see the symbolic link created from the temporary file to `/etc/passwd`:

Prog	R/EUID/RGID	Syscall	Arg1 (link)	Arg2 (target)	File1UID	File2UID
ln	1001/1001/1001	symlink	/etc/passwd	/tmp/.tmp.477	0	1001

[4]Here, the path identifier also contains the process ID, which is not shown, and the name of the process (`ssh`), which is shown.

[5]The path identifier for these events is derived from the names of the files involved.

Finally, we see the spool file (a link that ultimately expands to `/etc/passwd`, by now) being opened with the `O_CREAT` flags and write access:

Prog	R/EUID RGID	Syscall	Arg1 (file)	Arg2 (flags)	FileUID
lpr	1001/0 1001	open	/var/spool/dfA292	WRITE, TRUNC, CREAT	0

The presence of the `creat` flag also confirms the use of the older, deprecated `creat` system call, which, unlike the `open` call, does not re-check the file permissions before following the previously created and trusted symbolic link to `/etc/passwd`.

In addition to the lines that we have listed here, there are also extraneous `open` syscalls with `write` flags, but we ignore these since they do not involve `/etc/passwd` directly, nor are there symlinks between them and anything leading to `/etc/passwd`.

Therefore, the result of capturing links to or from files owned by `root` and all writes to files or links owned by `root` show us a clear and unambiguous path to rewriting the `passwd` file.

These results show the means of altering the password file (which could be any file owned by root, but the password file is one of the most critical on the system) as well as the actions that take place as a result of attaining them. Because the results also match the data indicated by the model, the results of this experiment validate the model for this exploit. And again, the results show that they do so with a relatively small amount of data. An important result of the experiment is also the observation about monitoring not only `open` calls but also `symlink` calls. Only by doing so can an analyst be certain of the actual target of the `open` call.

7.4 Avoid Authentication in su

The result of applying Analyze-Attack-Graph to *Avoid_Auth* (the procedure for monitoring avoidance of authentication is to capture executions of setuid root programs) in Section 6.4 is a directive to capture the the return value of the function that asks for passwords (the `pam_authenticate` library call on many modern UNIX-like systems), and any resulting programs that are spawned by the program avoiding authentication.

The results of recording these events for a version of `su`, that calls `pam_authenticate` but ignores the results, is as follows:

Prog	PID	R/EUID	RGID	Call	Return Value
su	693	1001/0	1001	execve	0
su	693			pam_authenticate	9
csh	694	0/0	0	execve	0
hostname	695	0/0	0	execve	0

The results in the table above[6] clearly show `su` being called and a root shell spawned, but they also show the return value of `pam_authenticate` being non-zero, which indicates failure. On the other hand, a version of `su` that spawns a root shell without even calling `pam_authenticate` entirely shows:

Prog	PID	R/EUID	RGID	Call	Return Value
`su`	706	1001/0	1001	execve	0
`csh`	707	0/0	0	execve	0
`hostname`	708	0/0	0	execve	0

In the table above, we see that a root shell is spawned by a normal user (uid = 1001) calling su, but `pam_authenticate` is *not* called, which should not happen. A root shell is being given "for free." What *should* occur for a normal, successful `su` is shown in this table:

Prog	PID	R/EUID	RGID	Call	Return Value
`su`	733	1001/0	1001	execve	0
`su`	733			`pam_authenticate`	0
`csh`	734	0/0	0	execve	0
`hostname`	735	0/0	0	execve	0

On the other hand, an unsuccessful `su` shows the results in the following table, which include an su and a failed `pam_authenticate` and also no spawned shell.

Prog	PID	R/EUID	RGID	Call	Return Value
`su`	855	1001/0	1001	execve	0
`su`	855			`pam_authenticate`	9

These results show the means of capturing a user password and proceeding without regard to whether the password actually matches the one that should be required for access to a particular resource. Because the results also match the data indicated by the model, the results of this experiment validate the model for this exploit. Again, the results show that they do so with a relatively small amount of data, consisting primarily of program executions and calls to the authentication library call.

7.5 Trojan Horse to gain root

The results of applying the ANALYZE-ATTACK-GROAL algorithm to the *Trojan_Horse* exploit that we built in Section 6.5 tells us that to monitor for Trojan horses, we need to monitor the execution search path. To view the execution search path used when launching a program, we could simply record the search path from a user's shell. However, this is a *state-based* approach

[6]The path identifier is derived from the PIDs of the processes, and the EUID of the first process.

to logging, as we discussed earlier. As an alternative, we can record the events which led to the program execution as a means of knowing the path. In the case of the `execve` system call, the execution path is searched serially, in order, to find the binary in question.

For example, assume the execution path is as follows:

```
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin
```

If a user calls `ls`, we see the following calls:

Prog	Syscall	Return Value
/sbin/ls	execve	2 (No such file or directory)
/bin/ls	execve	0 (Success)

Now, if the path is modified by an intruder so that a Trojan horse is called surreptitiously by adding the current working directory, we see:

```
PATH=.:/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin
```

Prog	Syscall	Return Value
./ls	execve	0 (Success)
evil_program	execve	0 (Success)

Therefore, we see that when the path is unmodified, `ls` shows up predictably in the `bin` directory and that version is executed. In the version with the modified path, `ls` shows up in the current working directory, is executed there, and is followed by an unknown process.

These results show the means of diverting a user's command to execute a program, so as to execute a different one. Because the results also match the data indicated by the model, the results of this experiment validate the model for this exploit. While the results do not show the exploit being installed, they do show the nature of the change, and the consequences, `evil_program`, of that change. Again, the results show that they do so with a relatively small amount of data, consisting primarily of program executions and therefore, recording of the execution search path.

7.6 Bypassing Standard Interfaces

The results of applying the ANALYZE-ATTACK-GROAL algorithm to the *Grab_TTY* and *Use_TTY* goals, describing the exploit of the world-writable `utmp` configuration flaw, which we described in Section 6.6, indicates that to monitor for methods of an attacker bypassing standard interfaces, a forensic system needs to monitor multiple sources of information and indicate when an inconsistency arises. Specifically, for the case of users and their associated terminals, as with the `utmp` file, the algorithm indicates the need to log the change in the ownership of a tty. This is done by recording writes to /etc/utmp and looking if one has changed ownership to root. In also indicates the need to record when other tty-related commands (system or library calls) monitor/read the tty:uid relationship. One historical example, in particular, is getlogin(). The key outcome of the algorithm is the need to reconcile data when multiple sources are supposed to contain the same data, but do not.

In modern FreeBSD variants, the /etc/utmp file is not group or world-writable. It is only writable by root. Further, the getlogin() call does not look in a file on disk for an association between users and terminals. It does this using a data structure inside the kernel's memory space. Therefore, we do not replicate the original utmp flaw exactly, but instead show a modern but closely related example of how a flaw could be detected by an attacker who may not have known how getlogin()'s source of data had changed. The attack graph and the results of applying the algorithm to the attack graph are the same.

For example, assume that /etc/umtp is world-writable. By recording file openings and writes, we can easily see the file being written. Also, since we record file openings, we can see the file later being read by a call to the UNIX command who. who correctly reads the data in the files, but given that the file has been tampered with, displays incorrect data. root is shown instead of sean:

Prog	Syscall	arg1	Return Value
vi	open	/var/run/utmp	7 (file handle)
vi	write	7 (file handle)	749 (bytes written)
who	execve	-m (myself)	0 (user ID = root)
who	open	/var/run/utmp	3 (file handle)

The incorrect information can also observed by monitoring the function call row from within the who program, that stores the information in a data structure:

```
root, ttyp0, Dec 2, 17:42, (72.132.232.36)
```

However, this deceiving result can be observed by instrumenting the libc interface to the getlogin system call, so that we observe the actual results of getlogin for the given user. The result is now shown as sean:

Prog	Syscall	Return Value
test	getlogin	sean

These results show the means of using a non-standard interface to alter a file deceive another program into acting based on the altered data. Because the results also match the data indicated by the model, the results of this experiment validate the model for this exploit. While it does not show the data being written, it does show the series of events, and the consequences, of that change. Again, the results show that they do so with a relatively small amount of data, consisting primarily of program executions and therefore, recording of the execution search path.

7.7 Summary of Experiments

These results of implementing several of the examples given in this chapter validate our forensic model by showing that when the data indicated by the model is recorded, the data helps an analyst understand, relatively easily, the nature of the attack, the method used, and the result. Further, in all cases, a relatively small amount of data is required to effectively analyze the intrusions.

8

Taking Laocoön from a Model to a System

Sherlock Holmes: *"At present I am, as you know, fairly busy, but I propose to devote my declining years to the composition of a textbook which shall focus the whole art of detection into one volume."*

—Sir Arthur Conan Doyle, "The Adventure of the Abbey Grange," *The Strand Magazine* (1897)

A portion of the work presented in this chapter was first published in an earlier paper by Bishop and Peisert [BP06].

Our current proof-of-concept implementation of Laocoön is an early but usable tool for analyzing a narrow class of exploits, right now. However, though we have largely established the model's effectiveness when applied to manually-drawn attack graphs, neither our model nor the implementation of the model are complete and final [PB07]. Our previous work in building a forensic model has left a number of open questions. Research into addressing these questions, as well as continued experiments along the lines of the work presented in this dissertation, will result in continued evolution of the model and its implementation. We expect the outcome will be continued improvements in efficiency, effectiveness, and eventually, expansion of the scope from computer hosts to networks and network devices.

8.1 Our Model in Practice

In the introduction, we mentioned there are four steps in using our forensic model-based approach:

1. Generate the attack graphs to be monitored. We describe a future approach to doing this in Section 8.2.

2. Instrument the system to collect the values described in the attack graphs. Some of the important, open issues are integrity and simultaneity.

3. Log the requested information during execution. Some of the important, open issues are execution overhead, space overhead, and the closely related problem of garbage collection.

4. Conduct a forensic analysis by reconstructing attack graphs with the logged data. This is done after an attack has been identified. Some of the important, open issues are the user interface and the problems arising from concurrent attacks.

The current Laocoön prototype addresses only a few of these issues, and even those in a preliminary manner. To to evaluate the efficacy and utility of Laocoön, we will address enough of these issues in the future to make the tool usable in an actual deployment. Once we have a robust enough tool, we will use it as a platform to test other ideas.

Efficiency of resource usage when the model is applied to a real system is difficult to measure, because in large part, the applications of the model are heavily situation-dependent. They are based on the attack graphs that are in place on the system in question, and on the use, both legitimate and illegitimate, that takes place on that system. Our current implementation results in a manageable amount of logged data and reasonable CPU slowdown. However, for practical purposes, the forensic system could be improved further, particularly to avoid attacks against the forensic system itself by causing many log messages and creating a denial-of-service. There are a number of methods that we could use to improve this while maintaining usability.

The instrumentations made to the FreeBSD kernel for the six experiments described in the previous chapter currently result in 28 MB/day of log data given a moderate amount of filesystem-oriented system executables and file editing tasks. On the other hand, given extremely intensive, constant compilation, our instrumentations generate 3.3 GB per day of raw, uncompressed, unoptimized forensic data, and a processor overhead of approximately 50%. However, these figures could be considered an upper bound on the data and processor requirements of implementing Laocoön for the examples that we have previously described, for a highly disk I/O intensive workload. In practice, we would expect these figures to be substantially lower. The reasons for this are first, at this point, we use no compression at all; and second, our rudimentary proof-of-concept implementation is focused on demonstrating the effectiveness of analyzing the attacks that we discussed in Chapter 7, not efficiency. As such, the implementation currently records *all of the forensic information* indicated in *all of our example attack graphs*, *all of the* *time*. In a practical implementation of the model, this would never be the case. A practical implementation would maintain enough state information to record only the data necessary for whatever attack graphs that an attacker appeared to be following, and subsequently, would record only the necessary *incremental* data. We now discuss future work to address some of these questions about efficiency.

8.1.1 Issues with Instrumentation

Instrumentation is largely an engineering issue. Some steps to improve the efficiency of our implementation are obvious. For example, data compression, even along the lines of `gzip` can often reduce the size of text files by 60-70%.[1] Another obvious step is to research the implications of performing periodic "garbage collection" and "throwing out" old data. Other steps are more complicated and involve more complicated research. We see the need for three enhancements: *enhanced enforcement*, *enhanced data capture*, and *enhanced data preservation*.

The addition of features such as NetBSD's *verified exec* [Lym04], which can force the use of instrumented compilers or designated binaries, is an example of *enhanced enforcement*.

Enhanced data capture refers to logging at intermediate layers of abstraction, such as tracing library calls, as well as providing markers to allow correlation of data at multiple levels of abstraction. For example, an attempt to alter the password file would cause log entries at the application layer, the library function layer, and the system call layer, and each particular action would have a unique tag that propagated from the highest layer of abstraction down to the lowest. That way, the analyst could instantly determine the exact system calls (for example) used to write data to a file, and then verify that the data passed to the related library function was in fact then passed to the system call without alteration.

This intermediate information, and the markers, would need to be descriptive and useful in forensic analysis. In addition to the logging mechanisms, pre-processing mechanisms, and tuning parameters available to a system administrator, a system designed with forensic principles in mind will *enhance data preservation* by, for example, using write-once media to store log information, or storing log information on a different system in a security domain distinct from the one being monitored. The logging mechanisms should be a reference monitor and reference validation mechanism to prevent the logging system from being bypassed or tampered with.

All of these principles have tradeoffs. Information may be obtained from logs, but using it requires several assumptions: that the information can be trusted, because it has not been tampered with [SK99]; that the programs being used logged any information at all (though most

[1]See gzip(1) man page.

system applications have some logging); and that the information is well-formed enough to be correlated [Bis95]. Information can also be derived from changes in data on the system. This is most useful when the needed information is not logged, or an attacker alters or deletes logs. An example is an attacker altering a password file on UNIX-like systems. On most systems, there will typically be no application-level log entry corresponding to this compromise. But by observing the data in the password file, an analyst can, under some conditions, immediately determine that the system is compromised.

Not every attacker is sufficiently sophisticated to alter or delete logs, so in some cases protecting the logs is unnecessary. But the simplicity of unprotected logs is balanced against the trustworthiness of those logs. Using write-once media prevents alterations, but requires replacement of the media. A "verified exec" feature could impede normal work at an installation, violating the principle of psychological acceptability for users [SS75]. A fault-tolerant stream of logged information [ZMAB03] that prevents execution when logging stops is subject to denial-of-service attacks.

Forensics involving concurrent sessions (for example, two attackers working simultaneously, either toward the same or different ends; or the same attacker with multiple, simultaneous logins) requires information about the (relative) order of events. We have not yet addressed this issue. This can be done by ensuring that events are recorded in the same order in which they happen. For example, race conditions are permitted but only if the order of events that describes the race is also recorded unambiguously. Further, if, for example, a system call's execution is interrupted by another event, the fact that the system call's execution was split into two, and the fact that an event occurred in between, needs to be recorded. Given how closely tied this issue is to low-level elements of a system, this area requires more work to evaluate the methods that could be used to implement this functionality and the efficiency of doing so.

At some point, the logging overhead can become large enough to make deterministic replay a desirable option [DKC+02]. This can be made inexpensive using hardware support [CFG+06, NPC05]. Replay in this context is done only during the last step of forensic analysis, and so it imposes very little during normal execution.

8.1.2 Issues with Logging

A practical implementation of Laocoön should have dynamic, active awareness of all currently-suspected attacks, and the current position of each of those attacks in each applicable attack graph. There can be multiple possible attack graphs to which a given attack might apply, with a non-zero number of nodes contained in the union minus the intersection of the graphs. For example, after a series of events, a given machine might be in a state that is shared among several different possible goals in several different attack graphs. This state information is contained in a part of our model called a *unique path identifier* (described in Chapter 5), and might appear as a set of unordered graphs and nodes, representing the possible positions the attacker could be in for each attack graph, as follows:

$$\{(graph_a, \{node_b, node_c\}), \ldots, (graph_d, \{node_e, node_f, node_g\})\}$$

We call this dynamic monitoring *active state awareness*. Given the overhead and criticality of its function, it would need to be implemented in a low-level element of the system (such as in the kernel or a protected routine, in a virtual machine, or on a separate coprocessor).

Using this active state awareness, one can reduce the amount of data necessary to log by using the unique path identifier to *log only the incremental data* necessary for the current set of possible nodes in the set of possible attack graphs with each state transition. By performing incremental data logging, much less data will be logged. The tradeoff is time: active state awareness increases the amount of processing time necessary and the memory footprint needed by the system to contain the representations of the attack graphs.

8.1.3 Issues with Forensic Analysis

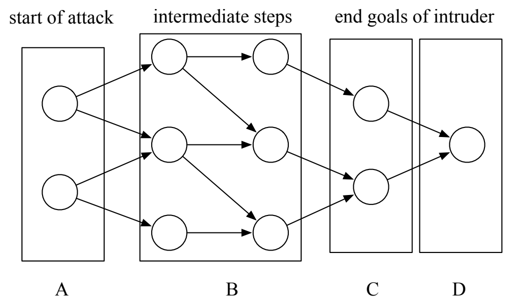
There are a number of ways to reduce the amount of data analyzed. The general idea is to use a tuning parameter to prioritize the data to examine, and thus, focus on particular paths in an attack graph. A number of metrics could aid in this simplification. For example, Manadhata and Wing [MW05, MW06] used *damage potential* and *effort* combined with the size of the *attack surface* of a system to create an *attack surface metric*. Our model could use this metric, or one based on the attack surface metric, that instead used the number of possible security policy violations from a given vantage point, rather than the number of possible vulnerabilities or the size of the "attack surface." The metric our model currently uses is *severity*.

Severity itself can be based on a variety of metrics. We currently use the level of access obtained:

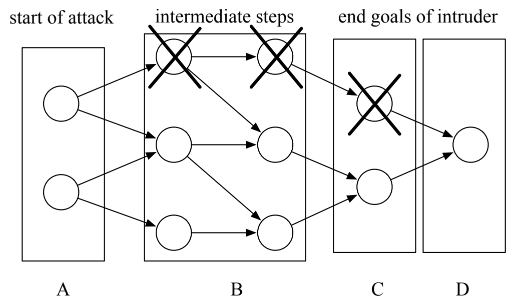
1. no access
2. ordinary user
3. obtaining partial root-level access, such as a file or process that *should* be owned by root (such as `/etc/passwd`), but is not, for some reason; or, convincing a particular process that a user is root, when they actually are not
4. obtaining root-level access

5. obtaining higher-than-root level access (such as single-user mode)

Other examples of severity could include using an ordering of the number of potential paths in an attack graph, where paths with fewer than n possible exploits, or paths longer than s steps could be placed low in the ordering. Additionally, in the next section, we discuss methods of using security policies to generate attack graphs. Therefore, another method of determining severity in the attack graphs would be to assign severities to each, specific security policy violation.



(a) A complete attack graph.



(b) The same attack graph, but with a path consisting of several goals unique to the path, and forenisc data unique to the goals eliminated.

Figure 8.1: Diagrams of two attack graphs, before and after path elimination has been applied.

If a forensic analyst finds evidence of a midpoint of a multistage attack, the analyst can determine from the attack graphs what the possible next (or previous) stages of the attack are, and look for evidence of those stages. For example, if an attack graph has three paths (see Figure 8.1(a)), and the evidence shows that the attacker could be following one of two paths, then only the information required to distinguish between the two paths need be examined. For forensic purposes, we can eliminate all data unique to the third path (see Figure 8.1(b)). This reduces the amount of data that must be analyzed, which is currently a serious problem in forensic analysis. If the detection occurs during the attack (say, using intrusion detection), the logging systems could be reconfigured to record only the information required to distinguish between the two paths. Note that the same intermediate goal may occur on multiple paths. So the analysis needs to isolate (possibly partial) sequences of goals and gather information to distinguish between different sequences. This suggests using artificial intelligence (AI) techniques, especially planning. AI techniques have been used in the past to automate forensic analysis of data, thus reducing the time a forensic expert needs to spend on manual analysis [SL03]. Probability of an attack occurring, perhaps based on the frequency of previous attacks, could also be used as a metric to eliminate potential paths, or reduce the amount of data collected for a particular set of paths.

Another useful heuristic for reducing the amount of logged data is statistical analysis. Suppose a particular set of paths involves a time interval of 100ms (perhaps a step exploiting a race condition). The attacker is unlikely to wait exactly 100ms between actions. But statistically, if the actions are separated by 95ms, or 125ms, it is reasonable to assume that the attacker tried to follow a path in that set; if the interval is 350ms or more, other paths are more likely.

Our experiments to date have results for analyzing a single attack at a time, using a single attack graph at a time. As we discussed earlier, it is important that the system also function with multiple attack graphs that can be analyzed simultaneously and automatically. But what happens to the efficiency of the automated analysis when the number of attack graphs grows? And how do attack graphs that share nodes in common get merged? We intend to research techniques to perform this process efficiently.

8.1.4 Issues with Construction

An obvious issue to be addressed is the user interface. The idea of a more useful human interface to forensic data has some precedent in existing solutions [CAS05, HWL95, KC05], though none of the solutions have been designed to work with a model like Laocoön. However, the model can be an asset rather than a liability because it can provide a means for uniformly describing and displaying data. Applying relational database concepts [GFM$^+$05, PBKM07a] to the information recorded by the model would also give powerful ways to formulate queries about the information. We will research methods of applying techniques from existing approaches, and will also research current human interface developments to determine what might be appropriate to use with Laocoön.

8.1.5 Legal Admissibility

Forensic analysis also refers to the derivation of information for use in court. We believe that the legal aspect of forensics can build on what we described in this dissertation, though the legal aspect was beyond the scope of this dissertation because we focused on the more general notion of collecting data to reconstruct events and activities. However, future work could add components to evaluate or enhance legal admissibility of digital evidence [SB03]. For example, by providing assurance of auditing and accountability of all actions taken within the software layer, then only the chain-of-custody of the hardware must be manually monitored. Currently, however, systems rarely have strong enough security and auditing features to support claims, beyond a reasonable doubt, that a system was used in a certain way or not.

8.2 Policy Discovery and Compilation

8.2.1 Introduction

Our model-based forensic approach requires attack graphs to drive the logging and forensic analysis. Thus, a key problem is generating attack graphs. So far, all of our attack graphs are generated by hand. We have used two different taxonomies of security flaws [ACD^{+}76, BH78] to direct our design of attack graphs. This has given us confidence in the value of our approach. But in a real deployment, attack graphs should be generated from system security policies. Doing so would allow one to make some kind of statement about the completeness of the monitored attacks. If security policies are not explicitly available (as is the case for most systems), they could be reverse-engineered from the system itself. A forensic analyst can then examine these generated policies, and change or augment them to more accurately reflect the actual policies that are expected to be followed for the system.

Attack graphs for real system policies can be large and complex. Hence, it is important to have the translation of policies to attack graphs automated. Doing so allows one to have more confidence in the soundness and completeness of the attack graphs than if they were generated by hand.

There are other ways of generating attack graphs. For example, one could do the equivalent of penetration analysis based on security flaw taxonomies. We do not consider such approaches in this section. Instead we will focus on the following methodology:

1. Reverse engineer security policies if they do not exist. Our approach is based on a description of system components and how their behavior is constrained by the system configuration. Examples of system configuration include the contents of configuration files, the permissions set on files, the existence of certain system objects (such as mutual exclusion locks or symbolic links to key files), and the versions of the installed software.

2. Translate the security policies into attack graphs, which are then used for goal-oriented, model-based forensic analysis.

An attack is a violation of a security policy. Such a violation can occur when a system is incorrectly configured, either by mistake or by design. This implies a gap, either known or unknown to the system administrator, between the security policy and the system configuration. We plan to bridge this gap. Once the gap is exposed, the system administrator can correct the configuration problem. In some cases, however, correcting the problem may overly constrain the behavior of the system.[2] In this case, the administrator can decide to use a generated attack graph to monitor for attacks that arise from the violation of security policy.

A different violation of security policy can arise from software and hardware faults. In this case, the gap (usually unknown) is between the expected behavior of the component and its actual behavior. We will also extend our techniques for configuration problems to software and hardware faults.

It is easy to see the value of reverse engineering security policies (which we call *policy discovery*). It allows the administrator to correct simple problems or explicitly decide to weaken the stated security policy to allow more behaviors. In addition, policy discovery can be used to drive intrusion detection by explicitly flagging cases in which the stronger security policy is violated.

The disconnect between a site's stated security policy and the policy actually enforced by the network and system configurations is often large. For example, an institution might ban the use of unencrypted connections, yet its servers might be configured to accept such connections. Conversely, an organization might require the use of specific protocols, and internal divisions might ignore the dictates due to fiscal or security constraints.

The gap becomes larger when a site has no stated security policy and decides to create one. The proper way to create such a policy is to ask each administrative sub-unit to determine what security requirements it has, and develop a policy that takes all these needs into account. The usual aproach is for the central administration to decide what a security policy should contain, and issue it, possibly creating procedures for exceptions in case some sub-units need latitude to perform their function. This can create problems, especially if the supplied policy is divorced from the reality of the work that the sub-units perform.

The situation for auditing compliance with a policy is worse. Auditors can make spot checks to determine if specific policy requirements are met, but such spot checks are complex. The reason lies in the complexity of modern systems. For example, a policy may require that no incoming `telnet` requests be accepted. The auditor checks that the internal hosts do not accept connections to port 23, and concludes that the policy is met. In reality, many internal hosts have a line in their configuration file to run `telnetd` when the system is rebooted, but the particular set of systems that the auditor checks may not happen to be running `telnetd`

[2]A computer left turned off and unplugged in a locked, windowless, airtight room, surrounded by a Faraday cage, can be considered secure but not terribly usable.

at that time. Alternatively, `telnetd` could have even been running on an alternate port. Had the auditor checked the configuration files, the policy violation would have been noticed.

Checking a system's configuration can expose the gaps between the actual policy and the configured policy. The analyst can then either alter the intended policy to reflect the actual policy, determine how to reconfigure the systems to implement the intended policy, or some of both.

Returning to our forensic model, Laocoön, we can not currently show in any formal manner that a set of data is minimal (*necessary*) for all implementations of a model, because there could be ways of analyzing the data to understand the attack that we, the designers of the attack graph, are unaware of. Given a complete enumeration and set of system configurations, though, we can formally validate a set of data as being the *necessary conditions* to analyze an attack, because those conditions can be shown to be the *only* ways in which policy violations can occur.

Such a process of unifying actual and intended policy falls short of our current abilities. We cannot express a policy in a language that policymakers can read, that also can be rigorously transformed into machine specific and operating system specific commands to implement the policy.

This section suggests the idea of languages encapsulating policies in both directions. We argue that not only must a higher-level policy language be transformable into an implementation of a policy (or set of policies) on systems, but also that it must be deriveable from the existing configuration of systems. This means that mechanisms to handle contradictions must be defined, as must syntax and semantics to handle procedures external to the computer.

We describe methodologies to enforce a policy or set of policies. This requires that the policy (or set of policies) be checked for consistency, systems and infrastructure be properly configured, human-oriented procedures provide the support that technological mechanisms cannot, and as the policy is updated to reflect changes in environment (such as new or evolving threats and laws or regulations), the changes are reflected in configurations. The element of *policy discovery* allows one to determine the actual policy of a new system being connected to the infrastructure, as well as to validate the enforcement of the existing policy or policies on systems being managed.

8.2.2 Background

Previous work in policy languages has focused on expressiveness of policy languages and the ability to transform policy languages into wrappers or other mechanisms that enforce policies.

The first type typically focuses on whether a policy language can express needed constraints, or which language is more powerful or appropriate for expressing particular constraints [CC97, JSS97, SZ97, ZS96]. The second typically allows a program to translate the policy statement into sets of configuration commands that will enforce the stated policy [BSSW96, DDLS01, HMT$^+$90, PH99]. Our work is closer to the second than the first.

Our work adds a layer of abstraction and the ability to "reverse engineer" the resulting configuration back into the policy language. Then we can apply techniques to analyze the results and determine soundness and completeness of the actual (as opposed to intended) policy. We can then iterate. The actual and intended policies are combined and checked for soundness and completeness. The analysts and policymakers deal with any resulting contradictions and undesireable aspects of the policies. Then the final policy is applied to the system.

Work on determining policy from existing configurations has focused on firewall rules. Typically these involve analyzing the firewall rules to create a database an administrator can query to determine what actions the rules allow [BMNW99, MWZ00, Woo01], or performing static or dynamic analysis of the rules to check for problems [ASH04, YMS$^+$06]. Previous work has also investigated extracting policy from logs [CGL00] and understanding the limits of on-line policy enforcement [Sch00]. Our work deals only with static discovery, and discusses abstracting configurations to an alternate, higher level of expression rather than focusing on handling queries, log entries, or execution monitoring.

8.2.3 Language Characteristics

Consider a policy language suitable for expressing policy constraints in a form that can be rigorously analyzed. Once the policy's soundness and completeness have been verified, the policy constraints determine how systems and infrastructure are to be configured. View the policy constraint language as a high-level programming language.

Taking this analogy further, the language must compile into an intermediate form in which the constraints are expressed in terms of services offered by the systems and infrastructure. While not focused on specific implementations, this level would specify (for example) what accesses file systems, web servers, and other components must provide. Think of this level as a bytecode-like language. We call this a "macro-assembly language." Then this form is assembled into machine-dependent, specific commands to configure individual systems and infrastructure components.

The advantages of this view to others are twofold. First, it provides a high degree of abstraction; if the infrastructure or details of systems change, the high-level policy constraints need not change; a simple recompilation (or reassembly) will provide the appropriate changes. The services are divorced from the details of their implementation, much as a macro assembly language abstracts the details of bit patterns from the machine language into macros. The second advantage is the explicit tie to reverse engineering policy. In this case, one can determine low-level (macro assembly) policy constraints for services by "disassembling" the configurations of systems, and then, ideally, "decompile" the server-based constraints into the higher level policy language. The result could then, for example, be compared to the (desired) policy constraints, and checked for inconsistencies.

Such a policy language must have several attributes. It must be extensible. Consider an electronic voting system. One county may forbid such a system to connect to any network when in use. Thus, the prototype language may have a simple construct that denies network accessibility. But other counties, in other states, may require that the voting system be connected to a telephone line, and call the county to send preliminary results to the central tabulation system. The security architects must be able to extend the language to describe this type of network (telephone line) connection as allowable [BBG07].[3]

As another example, if digital rights management is not included in the set of projects studied in our security architecture, it would need to be added. Perhaps specific components of digital rights management require language elements that are not present in the original language. The security management architects must be able to add those elements easily. The language must include procedural, as opposed to technical, elements. How to do this best is unclear (perhaps treat this element as a system, and generate specific rules much as the language mechanisms generate specific configuration rules for specific systems), but what is clear is that humans are central to any effective security system. A policy-based security architecture that does not take the human element into account is fatally flawed. Finally, the language must be able to capture environmental constraints. Security is a product of people, technology, and the environment in which people and technology must function. For example, the healthcare environment (where availability and confidentiality are paramount) has different security constraints than the financial environment (where integrity of bank accounts is paramount).

A policy constraint language also expresses a distributed computation, because the policy constraints might apply to many systems on a network, and the systems must work together to enforce the overall policy. For this purpose, policy changes that apply to more than one machine must occur simultaneously, for example if two machines control access to a

[3]While tempting to assert that all requirements should be known in advance, this is not always possible. In our election example, the state legislature and the county governments determine the laws, policies, and procedures; and as these are political issues, the governing bodies may change existing requirements or add new ones.

single resource (e.g. on a webfarm). Natural language policies must also translate to compatible configurations across multiple platforms. For example, if a policy requires encrypted remote logins, and one machine is only running ssh1, and another machine is only running ssh2, the policy enforcement mechanism must be able to re-configure one or both systems to be compatible with each other.

8.2.4 Overview of the Approach

Consider how a program expressed in a high-level programming language such as C or Pascal might be translated into a form that a computer can execute. First, a compiler transforms the high-level language into an intermediate form. The high-level language is (usually) machine independent. The statement "$x = 1 + y$" says nothing about how the variables "x" and "y" are represented internally, nor about how addition is managed. But the compiler translates this into an intermediate form that expresses the specific operations that must be performed. One such form is the abstract syntax tree (AST) that shows the variable "y" and the integer "1" under a node containing the operator "+", and that tree as a child of a node with "x" as the other child, and the node contains the "=" operator. This intermediate form is then translated into a machine-specific form—a machine language—that the target system can implement directly.

The intermediate form has two advantages and one disadvantage over the higher-level form. It abstracts the relationship of the components of the program. This allows the program to be analyzed in ways that the original form and the final (machine language) form do not allow. For example, the AST can be checked for type conflicts or analyzed to find dependencies on user input. Secondly, the AST is not tied to a particular system or indeed to a particular higher-level language. For example, the GNU compiler provides several front-ends for langauges such as C, C++, FORTRAN, and Ada. It uses the same code generation mechanism to generate code for any of these languages for a wide variety of systems, including Motorola 68000 systems, Intel x68 chips, and Sparc and MIPS-based systems. The disadvantage is the unreadability of the intermediate form. A programmer would have difficulty looking at an AST for a large routine and figuring out what it does, or programming in that form. The usual representation of a program is considerably easier to read.

We advocate a similar approach to expressing policies in policy languages. The high-level language should be designed with humans in mind, to ease the task of those writing policies (or translating policies from a natural language to the high-level language). A "policy compiler" would then transform this expression into an intermediate form, called the "intermediate policy representation" (IPRL). The IPRL is independent of system and infrastructure architecture, and expresses relationships in terms of services and rights. Then a "policy assembler" translates the IPRL into a set of configuration commands appropriate to a specific system.

This also eases the difficulty of reverse-engineering a policy from a specific system configuration. One first determines what services the system is configured to offer, and who may access them. This is translated into the IPRL. Then the IPRL can be "decompiled" into the higher-level policy language. Techniques of reverse-engineering applied in programming languages may be useful here (see for example [CG95, VO01]).

8.2.5 Applying Policies to Systems and Sites

The first step in enforcing a policy is "compiling" the policy into the intermediate policy representation (IPRL). The process of converting a policy to an IPRL involves a validation process, as well as building a hierarchical representation of the policy components, from the most general to the most specific.

A key aspect of this conversion is handling conflicts [MP06]. A hierarchy provides one way to resolve conflicts. Specifically, one can allow the more specific policies to violate the more general policies. For example, "no network access" could be overridden by "except responding to pings." Another means of resolving conflicts is to combine the relevant parts of the policies. For example, there should not be one policy which states that "only the superuser is able to execute a program" and another policy which says that "anyone can execute a program." Those policies would be in conflict. On the other hand, by specifying a *single* policy which states that "*both* the superuser and everyone else on the system can execute a program," a policy enforcement mechanism can be more certain that the policy is consistent and correct.

Validation must first be performed for internal consistency and next for consistency with the system to which it is applied. The validation for internal consistency is entirely machine-independent. There are many existing techniques for evaluating self-consistency of a policy, which includes making certain that there are no conflicting policies and that a policy is both syntactically and semantically correct. For example, this validation must be able to compare two sides of an assignment statement and be able to validate whether the right hand side is allowed to include a range of possible values, or must be binary (on or off, true or false, etc.).

The machine-dependent evaluation requires more specific domain knowledge. For example, the evaluation must make certain that specified subjects and objects exist on a given system, and that the policy's states and actions, both allowable and prohibited, are appropriate for the subjects and objects to which they apply. Consistency of a policy with a system requires that a policy be checked to ensure that the sub-policies are applicable to a designated system.

Some policies may be less precisely enforceable on some systems than others. For example, access control on traditional UNIX systems uses an abbreviated ACL, which is much less granular when compared with ACLs on modern implementations.

The IPRL does not possess domain knowledge, however, and thus must be written to deal with more than one possible system. The IPRL of policies like the one involving ACLs should be accompanied by an indication of how to handle situations in which ACLs are not implemented, and cannot be emulated precisely. A number of possible solutions exist. A warning could be generated, a more relaxed policy might be accepted, or an alternate solution could be found, which might include a different means of achieving similar goals on the system. Adding third-party software might compensate for functionality absent in the original operating system. If a more relaxed policy is to be accepted, then the IPRL must indicate the *minimum* acceptable policy as well as the *desired* policy.

Some policies may not be enforceable on some systems at all, either because the systems possess no means of ever violating the policy ("write" permissions on a file are irrelevant when the disk they are on is mounted "read-only") or because the systems possess no configuration mechanism with which to enforce the policy (if user home directories, temporary files, and swap space are required to be encrypted, but if there is no means of doing so, the policy is unenforceable). Many such situations may be unanticipated. In such cases, there should be default behavior that indicates what the policy enforcement mechanism should do. Regardless of whether the behavior is anticipated or the result of following defaults, one must specify whether the policy enforcement mechanisms should generate a warning and continue, or generate an error and halt.

We can view the translation of policies to system configurations by analogy, in much the same way that we view an ACL. ACLs are themselves simple policy grids which translate into permissions for accessing and manipulating system objects. In general, security policies also should be able to use a grid, tree, or lookup-table to map to a set of implementation mechanisms.

Once the IPRL has been mapped into the policy mechanisms on the particular type of system under consideration, the "compiler" can generate the actual configuration commands.

As an example of this scheme, suppose at the highest level, only users with logins are to be able to access systems. The IPRL might have a statement like:

```
if (login(user).allowed == NO)
      then system.access = NO;
else
      system.access = YES;
```

At a lower level, the “compiler” may be configured to check web and ftp services. The resulting code to generate the low-level, machine-specific statements from the IPRL might be as follows (comment lines, representing a pseudo-high-level policy language, begin with hyphens):

```
ftp:
   -- allow anonymous login if user
   --    need not have login to access system
   anonymous = (login(user).allowed == NO);
   ...

web:
   -- require password if user needs login to access system
   authentication = (login(user).allowed == NO);
   ...
```

The code to implement the low-level policy description as configurations on the system might translate the directives as follows:

```
echo ''ftp.anonymous = NO'' >> /etc/rc.local
echo ''allow any authenticated'' >> /usr/web/.configure
echo ''disallow all'' >> /usr/web/.configure
```

in which the first line explicitly turns off anonymous ftp, and the second and third force all web accesses to be authenticated for the particular target system.

8.2.6 Reverse-Engineering Policies

Policy discovery is the reverse of policy enforcement, and as such, many of the techniques are shared between both. Rather than starting at the policy and working down to the system, policy discovery starts at the system and works up to the policy.

A policy discovery process can reverse engineer policies in one of three ways: (1) it can examine the state of the system (e.g. filesystem, configuration files), (2) it can monitor events (or look at events previously logged), or (3) it can observe a system’s response to stimulus. The second method is very much like *vulnerability analysis* or *intrusion detection*. The third is similar to *penetration testing*. Some information can be gleaned in more than one way, and some information might need to be cross-checked for consistency. One of the concerns with combining method #1 with either #2 or #3 is there might be policy conflicts between them that are difficult to resolve.

Complicating matters is that an observed state is not static but *transient*. A transient state, as opposed to a fixed state, is one that changes over time. Unfortunately, by combining techniques, we may not be able to distinguish between static and transient events and states. Therefore, we cannot necessarily know whether we are viewing an actual policy or a deficiency in how a policy is implemented. Because implementation flaws are outside of the scope of this chapter, we do not consider discovery mechanisms that tell us about transient policies. Therefore, in the rest of this chapter, we focus on method #1: examining the state of the system that does not change, except by design.

Static policies can be reverse engineered from a finite number of key sources. For a BSD-like operating system, the key sources are:

1. Configuration files (including everything in `/etc` and `/usr/local/etc`). This includes user IDs and group IDs. Note that some of these files may be in users' home directories (such as `.rhosts` and `.shosts` files).

2. Ownership, locations, and permissions of files in the filesystem, including "special" files, such as device files.

3. Build configuration files, such as those for the kernel and other compiled binaries.

The most challenging part of policy discovery is assigning meaning to the data sources. While file ownership, location, and permissions are simple to represent and automatically process, configuration files or compiled binaries are significantly more difficult to process. This is another example of where analogy to source code compilation is useful. Binaries may actually require decompilation to fully understand, although if the build files are present, it is possible to extrapolate the policies built into the binaries from those files. For simple system configuration files, it may be sufficient to apply C++-like *templates* to subjects and objects. In doing so, a reverse-engineering process can distinguish subjects from objects and determine what is allowable in a particular context. For more complex system configuration files, a language specifying how to parse other languages might be needed. For example, regular expressions that indicate how to perform token lexing, and a grammar specifying productions (e.g. Backus-Naur Form (BNF)) with which to parse configuration files, might be needed. For example, the following configuration:

```
/etc/ssh/sshd_config:
PermitRootLogin no
```

might be read with this C++-style template:

```
/etc/ssh/sshd_config:
PermitRootLogin <boolean>
```

But this configuration:

```
/etc/master.passwd:
root:$1$SzebsWei$sjWoeMzpTjJkRVb:0:0::0:0:Charlie &:/root:/bin/csh
```

requires a more intricate regular expression, indicating optional fields and syntax of paths, and other fields in `/etc/master.passwd`. It should also be noted that not all configurations are relevant to security policy. The field in the file above containing "`Charlie &`" provides general information about the user and is irrelevant to the security policy.

As a more complex example, the following configuration potentially requires not only regular expressions to lexically analyze tokens, but a more complicated grammar to understand the hierarchical (and sometimes even recursive) productions. This configuration file controls access to the `root` account when a program called `sudo` [Mil] is used. That program verifies the user's identity by checking his or her password, and then grants access to other accounts based on the contents of the configuration file. The file below gives anyone access to all accounts.

```
/etc/sudoers:
ALL     ALL=(ALL) ALL
```

Once this and other configuration commands are decompiled into the IPRL, the result might contain:

```
login(root).remote = NO;
allowchangeEUID(root,ALLUSERS) = YES;
allowchangeEUID(root,ALLUSERS) = NO;
```

These policies demonstrate the need of the next step: once individual policy components generated by each configuration file have been checked, the components must be analyzed together, in a hierarchy, and checked for consistency. Discovered policies should not conflict. When they do, the resolution of the conflict depends (again) on domain-specific knowledge. For example, within one component, are the rules interpreted by the last match overriding the first, or the first match causing future matches to be ignored? Across components, what does having a non-empty password for the `root` account mean, when at the same time the `hosts.equiv` file is set to allow any remote user to log in as `root`? In the former, the actual policy must be discerned based on an analysis of the system components affected by the configuration file. In the latter, the actual policy has two components, one that bars access to `root` and the other that grants it to anyone. Perhaps other aspects of the system (such as a lack of servers that use the `hosts.equiv` file) provide the needed resolution.

Thus, the only way to resolve conflicts without having to generate an error and require human intervention is by assigning priority of policies using a hierarchy. For example, if “owner” and “world” permissions on a binary permit execution, but the “group” permissions do not, since “group” permissions are contained within “world” permissions in a hierarchy, we report that the policy allows anyone to execute the binary.

8.2.7 Detailed Example #1

In this section, we show examples of enforcing and reverse engineering policies. The examples involve enforcing and discovering policies related to a user’s ability to alter both their “real” and “effective” user IDs. The examples demonstrate how general the security policies need to be, and how specific the system configurations must be. For example, most systems have the concept of a “superuser,” but only UNIX-like operating systems call that user `root`. Therefore, the policy must reflect the more general concept. These examples also demonstrate a portion of the hierarchy of the machine-independent policy.

Enforcing Protected Superuser Access Suppose that we wish to impose a policy on a computer system that states that “a password is required for a user to perform any superuser functions on a computer system,” and that “remote access directly to any superuser privileges is prohibited.” An IPRL representation of the policy might be stated as simply as this (where again, a pseudo high-level policy language appears as comments beginning with hyphens):

```
system:
-- next line prevents remote logins as the superuser directly
login(superuser).remote = NO;
-- next line requires that the superuser account have a password
password(superuser).exists =  YES;
-- next line prevents any user from being able to execute
-- all programs as superuser
allowchangeEUID(superuser,ANYUSER,ALLCOMMANDS)  = NO;
-- next line prevents any user from executing a command to change
--   their effective user ID
allowchangeEUID(ANYUSER,ANYUSER,changeEUIDCommand) = NO;
```

On a machine-specific level, the configuration might be translated into these instructions on a UNIX-like system, shown in pseudocode:

```
grep '^root' /etc/master.passwd;
if (<password_field> is empty) {
    open(/etc/master.passwd);
    set root's password = ''xxxxxxxx'';
}
replace or append 'PermitRootLogin no' in /etc/ssh/sshd_config
function checkSudo () {
    /* This function parses the EBNF-formatted sudo grammar and
     *  removes any productions that allow any user to have sudo
     *  abilities (to root or any other user) on ALL commands.
     *  This function also specifically removes any function that
     *  permits a user to run the 'su' command as root, as well.
     */
}
```

The pseudocode instructions translate the policies into consistent, machine-specific configurations.

Reverse-Engineering Superuser Access To reverse-engineer policies relating to elevating user privileges on a system, we need to identify the relevant commands in the relevant files. First, we determine what network servers are available, because remote access requires the ability to connect to the system. In this case, we assume that machines are all UNIX-like, and that the remote secure shell, `sshd`, is the only server allowing remote logins. We can derive the IPRL from the configuration file as follows:

```
if sshd is running then:
      output ''RemoteAccess:'';
          find value of 'PermitRootLogin' in configuration file
                       /usr/ssh/sshd_config;
          output ''login(superuser).remote =
                       (PermitSuperuserLogin.value is yes)'';
endif
```

Next, we consider other configuration files that control access to `root`. Clearly, the password file is one because it contains a representation of `root`'s password. Another one is `sudo` as discussed above. For our purposes, we only consider these two. The appropriate IPRL can be generated as follows:

```
output ''login:'';
     find value of root's password in configuration file
                        /etc/master.passwd;
     output ''superuser_has_password = (password_field_length > 0)'';

output ''changeEUID:'';
     /* Parses the EBNF-formatted sudo (or equivalent) grammar
     *  and computes productions to determine their
     *  higher-level meanings;
     * return YES if any user can become superuser
     *
     */
     output ''can_execute_any_program(superuser) = '',
                        checkSudo(ANYCOMMAND);
```

Then taking this to the high-level language, we derive:

```
login(superuser).remote = NO;
password(superuser).exists =YES;
allowchangeEUID(superuser,ANYUSER,ALLCOMMANDS) = NO;
```

Suppose the first step discovers that "PermitRootLogin" is set to "no" in `sshd_config`, which results in the policy:

```
login(superuser).remote = NO
```

Then suppose the script discovers in the `/etc/master.passwd` file that the `root` password field is not empty, meaning `root` has a password. This results in the policy:

```
allowchangeEUID(superuser, ALLUSERS) = NO}
```

Finally, suppose that the `checkSudo()` function discovers in the `/etc/sudoers` file that sudo permissions allow any user to issue any command as `root`. This results in the policy:

```
allowchangeEUID(superuser, ALLUSERS, ALLCOMMANDS) = YES
```

Here, we have an inconsistency. Two policies, in the same place in the hierarchy, have conflicting settings. Given that they are in the same place in the hierarchy, we cannot resolve the intent of this configuration and must simply report both policies, the way in which they were derived, and the fact that a conflict exists.

8.2.8 Detailed Example #2

Consider the following case of a real exploit of the Network File System (NFS) [Sun89] that occurred at the San Diego Supercomputer Center (SDSC) at UC San Diego in 2004 [MB05, Sin05]. Indeed, a number of well-known exploits [Com94a, Com98b, van94, van98] have historically been used against NFS, as a result of implementation bugs and configuration errors. The key flaw exploited in this instance was that a configuration setting enabled NFS volumes to be mounted on "high ports," which are not restricted only to the superuser. Given this, and the fact that NFS-shared volumes can only limit the ability to mount a filesystem by IP address, and not by user, the filesystem could be mounted and manipulated by any user. Even if the NFS daemon had not allowed mounting on "high ports," the filesystem could still have been mounted on "low" ports by a user logged in as `root` on an untrusted system that was nevertheless in the range of IP addresses allowed by the NFS server. Indeed, there are reasons why such restrictions may not be desirable.

By virtue of the way NFS is designed, it is not possible to avoid all illicit access. It was simply not designed with high security in mind. However, the accesses and other operations can be logged, if our forensic model is applied to attack graphs based on violating policies that NFS is supposed to follow. At SDSC, a minimum of information was recorded, using standard forensic means: merely the IP address of the machine requesting an NFS mount and the time at which it occurred. But this is not enough information to analyze the attack.

This exploit arose because the file system could be mounted remotely by untrusted systems. This violated policy. An IPRL policy statement describing what is desired would be (comment lines, representing a pseudo-high-level policy language, begin with hyphens):

```
nfs:
-- only users in SUPERUSERS on hosts in TRUSTED or on networks listed
--    there can mount NFS file systems
SUPERUSERS = {"root"}
```

```
TRUSTED = { "a.ucsd.edu", ".sdsc.edu" };
files(fs).canmount(host,user) = host in TRUSTED, user in SUPERUSERS ;
```

This says that *user* on system *host* can mount file system *fs* only if that host is in the set *TRUSTED* and the user is in *SUPERUSERS*. This could translate into the following UNIX-based commands:

```
echo "$fs $host >> /etc/exports"
echo "secure >> /etc/exports"
```

which, when run on such a system, appends a line containing the name of the file system and the host that may mount it to the appropriate administrative file.

Assume the policy for NFS is as stated. To detect a vulnerability, generate the IPRL corresponding to the actual `/etc/exports` file. Suppose it is:

```
TRUSTED-rev = { "a.ucsd.edu", "b.sdsc.edu", "r.cs.ucsd.edu" };
files(fs).canmount(host) = host in TRUSTED-rev;
```

We now compose the two fragments of the IPRL and look for inconsistencies. In this case, the discrepancy between *TRUSTED* and *TRUSTED-rev* suggests an attack graph involving the host `r.cs.ucsd.edu` mounting a file system remotely.

8.2.9 Software/Hardware Issues

The problem of discrepancies between the actual policy and the intended policy is far more difficult to analyze when the discrepancies are introduced because of faulty software or hardware. The problem is that the faults are unknown. Developing techniques to analyze software for vulnerabilities is the focus of much ongoing research.[4] Rather than duplicate it, we build on the results of that research by focusing on ways to incorporate the results into this project.

As programs depend on system environments, they in effect play the role of system-level configuration data. Suppose a privilege-enhancing program (like the `su` program) has a flaw in its authentication mechanism that allows anyone to acquire a particular right. That may be treated as a misconfiguration in which a system database of privileges has that right added to every user. The problem is to determine *which* privileges — or more generally, configuration errors — each program corresponds to. Given the flaws in a program, the "configuration errors"

[4]Such as *property-based testing* [FB97].

can be derived. *How* to do so is another matter, leading to the question of how to translate from flaws to equivalent configuration errors.

Hence the specific research question that we will explore is how to translate the *results* of software (and hardware) analysis tools to appropriate configuration representations. This will require an analysis of the types of flaws in programs and thus build on the results of prior research. It also requires that we develop appropriate policy language constructs to represent the problems, and integrate them into the IPRL described above. To demonstrate the viability of the approach, we will take the output of several (static and dynamic) analysis tools and show how to do the translations.

8.2.10 Procedural Policies

Not all "good security practices" can be translated into policies. This is particularly true where human procedures are involved, and when the "good security practices" are too general for a language to be able to ultimately make them machine-specific. For example, a principle of "good forensic analysis" is to "consider the entire system, including the user space, in logging." [PBKM05] This kind of principle cannot be easily enforced, because the policy is too general to have any way of translating properly into an intermediate language.

On the other hand, the policy could state that state that function calls must be captured [PBKM07a]. On UNIX, the policy could translate into a system-level mechanism that looks to see whether the `exec` system call has been modified to force all binaries on the system to be run through a specific dynamic instrumentation tool that records function calls.

Not all policies can be easily enforced by a computer system, even if they can be specified using the policy language. For example, if a policy states that a country requires that all cryptographic keys be registered with the police [Bis05a], there is no easy way of this short of augmenting the system to contain capabilities that it did not originally have. That is, the system would transmit all of the keys to the police.

The question of how to handle procedural aspects of policy affects the usefulness of the high-level language. Take our above example of no password for the superuser account. If the high-level language mandates that there be one, it must also mandate that the password not be removed (or provide explicit circumstances under which it may be removed). In this case, the compiler must generate some procedural instructions to support this (implicit) requirement.

One way to do this is to consider the procedural practices to be another system. Then, just as the IPRL can be translated to a particular set of configuration commands for a Windows system, and a different set for a Linux system, so could it be translated to a set of instructions (or policy manual references) for humans to follow. So translation of the policy to procedural practices is similar to the generation of configuration information from the IPRL.

But reverse-engineering procedural practices is more complicated. Ultimately, one needs to put them into a form that can be translated back to the IPRL. The development of such a form, and a demonstration of its completeness (so that any relevant procedural practices can be entered), is an area for future work.

8.2.11 Sufficiency and Necessity in the Forensic Model

In the future, we will research methods for understanding and proving the sufficient and necessary data in the general case, for any attack graph. We discuss the intuition for these now:

sufficiency $G1$ and $G2$ represent attack graphs from the set of attack graphs that define the universe of security violations. If a sequence is accepted by $G1$, the interpretation is that the information is sufficient to identify $G1$ as an attack graph potentially representing the actual attack. So if it's accepted by $G1$ and $G2$, it's *sufficient* to identify multiple potential attack graphs.

But each attack graph is an FSA, and thus can also be thought of as a predicate that is true exactly for the sequences it accepts. So, "sufficiency" means that each attack graph is specific enough to uniquely identify the violation, which means that no more than one attack predicate is true at any time.

More formally, given two state machines $G1$ and $G2$, is there a sequence s that is accepted by both? In logical, mathematical notation: $\neg G1 \vee \neg G2$.

We will first research whether our intuition on generalizing the method of determining *sufficient* data for analyzing attacks is correct, and then research methods of proving it.

necessity Necessity is harder to prove without a formal specification of the system. Let S be all the valid behaviors of the system. That is, a behavior that includes security violation is not a sequence in S. Consider an attack graph G. Since G is a valid attack graph, no sequence in S is accepted by G.

Consider another attack graph G' that uses fewer variables than G and such that all sequences accepted by G are also accepted by G'. it's easy to construct such a G'; it's just a weakening of G. A trivial G' is "true" (the attack graph that accepts all sequences). But, there's no obvious reason why G' would continue to exclude sequences in S, and so one would need to have S available to check whether G' is a bonifide attack graph.

Using logic, we have $G \Rightarrow \neg S$. We want the weakest predicate $G' : G \Rightarrow G'$ where $G' \Rightarrow \neg S$.

8.2.12 Conclusions

We have described methods for enforcing security policies by translating them to machine-dependent configurations, via an intermediate, machine-independent policy representation. We have also described methods for "discovering" policies by reverse-engineering them from static system configurations, again via a machine-independent, intermediate representation. This bi-directional language translation helps to unify the difficult and often distinct tasks of specifying policies, enforcing them, adding new systems to a network, and verifying policy compliance.

By looking only at static system configurations, we are ignoring vulnerabilities — flaws in the *implementation* of policy — but it is not our goal to prove, or even determine, that a running operating system, let alone a network of systems, is secure. Rather, we have described a method to enforce a given policy properly, and to discover the actual policy that is being enforced. Likewise, our methods help bring *misconfigurations* to light.

Not all policies can be enforced as easily as modifying a configuration file, or modifying permissions of a file. For example, we may wish to disallow some users from executing setuid/setgid programs, to prevent them changing their effective UIDs. Without the advantage of detailed ACLs, this may be non-trivial. Additionally, some changes to the system based on policies being enforced may require recompiling binaries (or the kernel) with different configuration parameters, and even rebooting, for the changes to be reflected in running system. Also, some *actual* configuration files are machine-generated by system programs, based on human-generated, configuration-file counterparts.

This methodology may help people and institutions to determine what their system allows and prohibits. System software developers could use this approach to help users resolve conflicts when configuring their software. This would ensure a more stable system that meets the intent of the policy, rather than one that unintentionally drifts into a new, less desireable policy, thereby compromising the institution.

8.3 Applying Forensic Techniques to Intrusion Detection

The overlap between forensic analysis and intrusion detection lies in the analysis of the attacker's actions. Intrusion detection looks for an attacker; forensic analysis determines *how* an attack occurred, *what* the attacker did, and *what effects* the attack had. The information revealing that an attack has occurred also reveals some information about how the attack occurred, and thus what the effects of the attack were. Thus, techniques developed for forensic analysis can be applied to systems for which the existence of an attack is undetermined.

In order for the intrusion detection system to determine if the system is under attack (or has been attacked), it must analyze information to detect the attack. Forensic analysis uses information obtained from system and network logs to reconstruct what happened and/or the state of the system at a particular point in time.

As our work in forensic analysis using sequences of function calls shows (Chapter 3 [PBKM07a]), intrusion detection techniques can benefit forensics by helping to locate the events to analyze. This is because the problem of computer forensics is not simply finding a needle in a haystack: it is finding a needle in a stack of needles. Given a suspicion that a break-in or some other "bad" thing has occurred, a forensic analyst needs to localize the damage and determine how the system was compromised. With a needle in a haystack, the needle is a distinct object. In forensics, the point at which the attacker entered the system can be very hard to ascertain, because in audit logs, "bad" events rarely stand out from "good" ones.

Conversely, because of the overlap between the two fields, our work in forensic analysis can also be used to enhance intrusion detection. For example, new signatures for intrusion detection could be built after careful forensic analysis. Also, using anomaly detection techniques in a *post mortem* fashion allows an analyst to find rare or missing events, including the absence of expected events, as well as the presence of unexpected events. This enables an analyst to discover when an event that should occur does not. We intend to investigate methods of exploring the benefits of cross-fertilization among these two fields further:

As an example, earlier in this chapter, we discussed a number of ways in which probabilistic and statistical metrics could enhance the practicality of our forensic model by limiting the data that needs to be collected to perform forensics, or at least highlight it for the analyst based on the most likely or most damaging scenarios. Many of these techniques have been used in intrusion detection systems, particularly anomaly-based systems [TC90], and by combining the results of these systems with our model, we may be able to generate a likely set of attack graphs automatically.

The other key area we will examine is how a system, using attack graphs built by policy discovery and analyzed using our forensic model, can be applied directly to intrusion detection. Our model already collects data about attack goals as they occur, but the real-time analysis is limited. By performing more real-time analysis that leverages the unique path identifier (described earlier) to correlate events, and using probabilistic assumptions about the nature of attacks and other statistical metrics, the system could become much more useful as an intrusion detection tool. This is because intrusion detection is ideally automated, on-line, and real-time. The goal is to detect an intrusion as quickly as possible so that the damage can be contained. Though the necessary processing time would increase with this kind of real-time analysis, the data storage requirements could decrease dramatically. In many cases, attacker goals in attack graphs could simply be "checked off" as having occurred. In other cases, the logging data might be correlated, summarized, or otherwise compressed, rather than forcing the system to record enough data to perform the most rigorous possible analysis.

A portion of the work presented in this chapter was first described in an earlier paper, "Your Security Policy is *What??*" Matt Bishop and Sean Peisert, *University of Calfornia at Davis Technical Report*, CSE-2006-20, October 2006.

9

Conclusions

SHERLOCK HOLMES: *"You see, my dear Watson" — he propped his test-tube in the rack, and began to lecture with the air of a professor addressing his class — "it is not really difficult to construct a series of inferences, each dependent upon its predecessor and each simple in itself. If, after doing so, one simply knocks out all the central inferences and presents one's audience with the starting-point and the conclusion, one may produce a startling, though possibly a meretricious, effect. Now, it was not really difficult, by an inspection of the groove between your left forefinger and thumb, to feel sure that you did* not *propose to invest your small capital in the goldfields. . . . Every problem becomes very childish when once it is explained to you."*

—Sir Arthur Conan Doyle, "The Adventure of the Dancing Men,"
The Strand Magazine (1903)

9.1 Summary

This dissertation presents a rigorous, goal-oriented forensic model, called Laocoön, that when applied to a computer system, logs data that is significantly more effective than previous approaches to forensic logging and analysis, at acceptable cost in processor overhead and disk space. The model we have presented is inspired by a set of qualities to enhance forensic models, which we have also presented. Likewise, those qualities are derived from a number of principles of forensic analysis, which we also derived from the failings of existing approaches.

Laocoön builds upon the existing requires/provides model. By inverting that model and applying a set of functions and algorithms to a set of key attack graphs, starting with the end of the attack graph (an intruder's "ultimate goal"), we have presented a methodology for understanding the data necessary to record in order to do a thorough forensic analysis, and also for understanding the ways in which a system needs to be instrumented to capture that data.

This dissertation also shows examples of successfully applying Laocoön. The application of Laocoön to the examples — which cover the entire space of two well-established lists of flaw domains — and the analyses that show the data indicated by the model is *necessary*, gives us confidence in the value of our approach. Finally, the results of implementing several of the examples and the analysis of that resulting data, which shows that the data is *sufficient*, validates the model. The results show that a comparatively small amount of data is required to analyze the intrusions effectively.

Ideally, we would like to be able to directly and formally compare both the effectiveness and efficiency of our methods to existing methods. However, as we discussed in Section 8.1, the efficiency of applying the model is difficult to measure. Similarly, for a measure of effectiveness, despite its significant flaws, there is no forensic equivalent even to the Lincoln Labs' network intrusion detection datasets [McH00] to compare our techniques against. A measure of effectiveness would probably be based largely on experiments with humans, and, as far as we are aware, no such previous, standardized experiments to compare against have been published.

Nonetheless, we can state that unlike existing work, including *our* previous ad hoc approach, we have shown a method that logs the necessary data to perform forensic analysis without much duplication. For example, we can conceptually compare the use of the model to the existing "toolbox" approach, as well as our sequences of function calls method: with our model, we see a clear, unambiguous path reflecting the intruder's goals and the methods used to achieve them. With the toolbox approach, we see a great deal of overlapping data in some areas (for example, output from a network intrusion detection system, in addition to log messages from TCPWrappers), and large gaps in data in other places. With the function calls method, we showed the differences between normal operation and anomalous operation, but that difference may or may not describe the origins, methods, exploit, or consequences in the necessary detail.

Therefore, we can say that the implementation of Laocoön outputs both *sufficient* and *necessary* data to forensically analyze intrusions, and does so in a way that provides an analyst with a much higher degree of confidence in the accuracy and completeness of the analysis than prior approaches to computer forensic analysis.

9.2 Recommendations

Our model is a piece in the puzzle that describes a relationship between data needed to analyze and understand events, and the events themselves. Additional work is required to automatically generate attack graphs and capability pairs for the goals in the graphs, and to translate the formalizations of the goals into information to log. This future work will continue to improve and help validate our research. Nevertheless, we have shown techniques that can be used to derive the information necessary for a human to perform forensic analysis without logging universally impractical amounts of data.

While current operating systems lack well-structured forensic capabilities, new systems should be capable of fostering effective forensic analysis at minimum cost. We believe that a system using this forensic model should be much more successful at both efficiently and effectively performing forensic analysis. Indeed, we believe that after a sufficient number of the enhancements described in Chapter 8 have been made — such as those to efficiency, active state awareness, and protection of the logging mechanism — that Laocoön could be integrated into the kernel of an operating system such as Linux or BSD, and used by forensic analysts with a high degree of success. A broader (but not necessarily less detailed) perspective might also be gained from also instrumenting network devices — such as switches — with the model, and correlating data between machines. This would also require that attack graphs being employed include networks and network devices. Alternatively, new systems, particularly those with limited purpose and functionality, such as the "supervisor" system at a voting site that controls the electronic voting machines themselves, could be completely redesigned using Laocoön and our other forensic principles, with rigorous forensic logging and auditing in the design.

While the average user of a system will not be capable of using the output of Laocoön to perform a forensic analysis themselves (and perhaps might never be able to do so, even with dramatic human interface enhancements), there are enough critical systems and users connecting to critical systems from relatively insecure systems, that it is inevitable that forensic analysis of almost any type of system will be desired. Having the proper data available for a forensic analyst from the implementation of a forensic model such as Laocoön could enable complete and accurate analysis on *any* system that it is installed on, when needed.

Bibliography

Dr. Jones (Jr.): *"Seventy percent of all archaeology is done in the library. Research. Reading."*

Indiana Jones and the Last Crusade (1989)

[8LG] 8LGM. UNIX `lpr` security advisory. `http://www.8lgm.org/advisories/[8lgm]-Advisory-3.UNIX.lpr.19-Aug-1991.html`.

[ACD+76] Robert P. Abbott, J.S. Chin, James E. Donnelly, W.L. Konigsford, S. Tokubo, and D. A. Webb. Security Analysis and Enhancements of Computer Operating Systems (RISOS). Technical report, Lawrence Livermore Laboratory, April 1976.

[Ale96] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), 1996.

[All05] Eric Allman. Personal conversations, January 2005.

[And72] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Vol. II, ESD/AFSC, Hanscom AFB, Bedford, MA, Oct. 1972.

[And80] James P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, April, 1980.

[ASH04] Ehab Al-Shaer and Hazem Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *Proceedings of IEEE Infocomm*, March 2004.

[Bac00] Rebecca Gurley Bace. *Intrusion Detection.* Macmillan Technical Publishing, 2000.

[BBG07] Eric Barr, Matt Bishop, and Mark Gondree. Fixing the 2006 Federal Voting Standards. *to appear in the Communications of the ACM*, 50(3), March 2007.

[BCF+97] Matt Bishop, Steven Cheung, Jeremy Frank, James Hoagland, Steven Samorodin, and Chris Wee. The Threat from the Net. *IEEE Spectrum*, 34(8):56–63, August 1997.

[BD96] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, 1996.

[BH78] Richard Bisbey and Dennis Hollingworth. Protection Analysis: Final Report (PA). Technical report, Information Sciences Institute, May 1978.

[Bis83] Matt Bishop. Security Problems with the UNIX Operating System. unpublished, 1983.

[Bis86] Matt Bishop. How to Write a Setuid Program. *;login:*, 12(1), January/February 1986.

[Bis87] Matt Bishop. Profiling Under UNIX by Patching. *Software–Practice and Experience*, 17(10):729–740, October 1987.

[Bis88a] Matt Bishop. Auditing Files on a Network of UNIX Machines. In *Proceedings of the USENIX UNIX Security Workshop*, pages 51–52, August 1988.

[Bis88b] Matt Bishop. A Fast Version of the DES and a Password Encryption Algorithm. Technical Report RIACS 87.18, Research for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, July 1987 (Revised August 1988).

[Bis89] Matt Bishop. A Model of Security Monitoring. In *Proceedings of the Fifth Annual Computer Security Applications Conference (ACSAC)*, pages 46–52, Tucson, AZ, December 1989.

[Bis95] Matt Bishop. A Standard Audit Trail Format. In *Proceedings of the Eighteenth National Information Systems Security Conference*, pages 136–145, October 1995.

[Bis99] Matt Bishop. Vulnerabilities Analysis. In *Proceedings of the Second International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 125–136, September 1999.

[Bis02] Matt Bishop. *How Attackers Break Programs, and How to Write Programs More Securely.* SANS 2002, Baltimore, MD, May 2002.

[Bis03] Matt Bishop. *Computer Security: Art and Science.* Addison-Wesley Professional, Boston, MA, 2003.

[Bis05a] Matt Bishop. The Insider Problem Revisited. In *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, pages 75–76, Lake Arrowhead, CA, October 20–23, 2005.

[Bis05b] Matt Bishop. Position: "Insider" is Relative. In *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, pages 77–78, Lake Arrowhead, CA, October 20–23, 2005.

[BL75] David Elliott Bell and Leonard J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report EST-TR-75-306, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, 1975.

[BMNW99] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A Novel Firewall Management Toolkit. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[BN89] David F.C. Brewer and Michael J. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989.

[Bon80] David Bonyun. The Role of a Well-Defined Auditing Process in the Enforcement of Privacy and Data Security. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy*, 1980.

[BP06] Matt Bishop and Sean Peisert. Your Security Policy is *What*?? Technical Report CSE-2006-20, University of California at Davis, 2006.

[BPWC90] J.L. Berger, J. Picciotto, J.P.L. Woodward, and P.T Cummings. Compartmented Mode Workstation: Prototype Highlights. *IEEE Transactions on Software Engineering*, 16(6):608–618, June 1990.

[BSSW96] Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A Domain and Type Enforcement UNIX Prototype. *Computing Systems*, 9(1), Winter 1996.

[Car05] Caleb Carr. *The Italian Secretary: A Further Adventure Of Sherlock Holmes.* Caroll & Graf, New York, NY, 2005.

[CAS05] Gregory Conti, Mustaque Ahamad, and John Stasko. Attacking Information Visualization System Usability Overloading and Deceiving the Human. In *2005 Symposium on Usable Privacy and Security (SOUPS)*, 2005.

[CC97] Laurence Cholvy and Frederic Cuppens. Analyzing Consistency of Security Policies. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 103–112, Oakland, CA, May 1997.

[CFG$^+$87] P. T. Cummings, D. A. Fullan, M. J. Goldstien, M. J. Gosse, J. Picciotto, J. P. L. Woodward, and J. Wynn. Compartmented Model Workstation: Results Through Prototyping. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 1987.

[CFG$^+$06] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-Based Architectures for General-Purpose Monitoring of Deployed Code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Security (ASID)*, October, 21 2006.

[CG95] Christina Cifuentes and K. John Gough. Decompilation of Binary Programs. *Software—Practice and Experience*, 25(7):811–829, July 1995.

[CGL00] Christina Yip Chung, Michael Gertz, and Karl Levitt. Discovery of Multi-Level Security Policies. In *Proceedings of IFIP Workshop on Database Security (DBSec)*, pages 173–184, 2000.

[Che92] Bill Cheswick. An evening with Berferd in which a cracker is lured, endured, and studied. In *Proceedings of the Winter USENIX Conference*, January 1992.

[CLF03] Steven Cheung, Ulf Lindqvist, and Martin W. Fong. Modeling Multistep Cyber Attacks for Scenario Recognition. In *Proceedings of the Third DARPA Information Survivability Conference and Exposition (DISCEX III)*, volume I, pages 284–292, Washington, D.C., April 22–24 2003.

[CM02] Frédéric Cuppens and Alexandre Miège. Alert Correlation in a Cooperative Intrusion Detection Framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, CA, 2002.

[Com94a] Computer Emergency Response Team. NFS Vulnerabilities. CERT Advisory CA-94.15, December 1994.

[Com94b] Computer Emergency Response Team. Writable /etc/utmp Vulnerability. CERT Advisory CA-94.06, March 1994.

[Com98a] Computer Emergency Response Team. IP Denial-of-Service Attacks. CERT Advisory CA-97.28, December 1997, Revised May 1998.

[Com98b] Computer Emergency Response Team. Remotely Exploitable Buffer Overflow Vulnerability in mountd. CERT Advisory CA-98.12, October 1998.

[Com99a] Computer Emergency Response Team. Buffer Overflow in Sun Solstice AdminSuite Daemon sadmind. CERT Advisory CA-99.16, December 1999.

[Com99b] Computer Emergency Response Team. Buffer Overflows in SSH Daemon. CERT Advisory CA-99.15, December 1999.

[Com99c] Computer Emergency Response Team. Trojan Horse Version of TCP Wrappers. CERT Advisory CA-99.01, January 1999.

[Com00] Computer Emergency Response Team. Multiple Buffer Overflows in Kerberos Authenticated Services. CERT Advisory CA-2000.06, May 2000.

[Com01] Computer Emergency Response Team. Nimda Worm. CERT Advisory CA-2001.17, September 2001.

[CW87] David D. Clark and David R. Wilson. A Comparison of Commercial and Military Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. In *Proceedings of Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[Den87] Dorothy E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[DKC$^+$02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. 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)*, 2002.

[ER89] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Oakland, CA, 1989.

[FB97] George Fink and Matt Bishop. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, July 1997.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, 1996.

[fid01] fides. Simple Buffer-Overflow Exploits. *Collusion E-zine*, 23, May 2001.

[Fre98] FreeBSD, Inc. FreeBSD Security Advisory: LAND Attack Can Cause Harm to Running FreeBSD Systems. FreeBSD-SA-98:01.land, January 1998.

[FV] Dan Farmer and Wietse Venema. The Coroners Toolkit (TCT). `http://www.porcupine.org/forensics/tct.html`.

[FV04] Dan Farmer and Wietse Venema. *Forensic Discovery*. Addison Wesley Professional, 2004.

[GFM+05] Ashvin Goel, Wu-chang Feng, David Maier, Wu-chi Feng, and Jonathan Walpole. Forensix: A Robust, High-Performance Reconstruction System. In *Proceedings of the 25th International Conference on Distributed Computing Systems Workshops*, pages 155–162, 2005.

[GR03] 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)*, 2003.

[Gro97] Andrew H. Gross. *Analyzing Computer Intrusions*. PhD thesis, University of California, San Diego, Department of Electrical and Computer Engineering, 1997.

[GWSB03] David P. Gilliam, Thomas L. Wolfe, Josef S. Sherif, and Matt Bishop. Software Security Checklist for the Software Life Cycle. In *Proceedings of the Twelfth IEEE International Workshop on Enabling Technologies: Infrastructure for Colaborative Enterprises (WETICE'03)*, 2003.

[HB96] L. Todd Heberlein and Matt Bishop. Attack Class: Address Spoofing. In *Proceedings of the 19th National Information Systems Security Conference*, pages 371–377, October 1996.

[HFS99] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*, 6:151–180, 1999.

[HMT+90] Allan Heydon, Mark W. Maimone, J.D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miro: Visual Specification of Security. *IEEE Transactions on Software Engineering*, 16(10):1185–1197, October 1990.

[HWL95] James A. Hoagland, Christopher Wee, and Karl N. Levitt. Audit log analysis using the Visual Audit Browser. Technical Report CSE-95-11, University of California, Davis, 1995.

[JKDC05] Alshlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrustions through Vulnerability-Specific Predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles*, 2005.

[JLS76] Anita K. Jones, Richard J. Lipton, and Lawrence Snyder. A Linear-Time Algorithm for Deciding Security. In *Proceedings of the 17th Symposium on the Foundations of Computer Science*, pages 33–41, October 1976.

[JSS97] Sushil Jajodia, Pierangela Samarati, and VS Subramanian. A Logical Language for Expressing Authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[KC05] Samuel T. King and Peter M. Chen. Backtracking Intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, February 2005.

[KMLC05] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching Intrusion Alerts Through Multi-Host Causality. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[KS94] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 1994 ACM Conference on Communications and Computer Security (CCS)*, November 1994.

[Kup04] Benjamin A. Kuperman. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources.* PhD thesis, Purdue University, 2004.

[LCM$^+$05] 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 Programming Language Design and Implementation (PLDI)*, June, 2005.

[Lym04] Brett Lymn. Verified Exec - Extending the Security Perimeter. In *Linux.Conf.Au*, 2004.

[m3l] m3lt. The LAND attack (IP DOS). `http://www.insecure.org/sploits/land.ip.DOS.html`.

[MB05] John Markoff and Lowell Bergman. Internet Attack is Called Broad and Long Lasting. *New York Times*, Late Edition - Final, Section A, Page 1, Column 1, May 10 2005.

[McH00] John McHugh. Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by the Lincoln Laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):262–294, November 2000.

[Mil] Todd Miller. `sudo` (“superuser do”). `http://www.sudo.ws/`.

[MMDD02] Benjamin Morin, Ludovic Mé, Hervé Debar, and Mireille Ducassé. M2D2: A Formal Data Model for IDS Alert Correlation. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 115–137, 2002.

[MP06] Patrick McDaniel and Atul Prakash. Methods and Limitations of Security Policy Reconciliation. *ACM Transactions on Information and System Security (TISSEC)*, 9(3):259–291, August 2006.

[MSB$^+$06] David Moore, Colleen Shannon, Doug J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, May 2006.

[MVVK06] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous System Call Detection. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):61–93, February 2006.

[MW05] Pratyusa K. Manadhata and Jeannette M. Wing. An Attack Surface Metric. Technical Report CMU-CS-05-155, Carnegie Mellon University, July 2005.

[MW06] Pratyusa K. Manadhata and Jeannette M. Wing. An Attack Surface Metric (position paper). In *Proceedings of the First Workshop on Security Metrics*, August 2006.

[MWZ00] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A Firewall Analsis Engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Oakland, CA, 2000.

[NAS] NASA. The Vulnerability Matrix of Top 50 Security Problems. `http://nob.cs.ucdavis.edu/testing/papersetc/vulner.html`.

[NB06] Vicentiu Neagoe and Matt Bishop. Inconsistency in Deception for Defense. In *Proceedings of the 2006 New Security Paradigms Workshop (NSPW)*, 2006.

[NCRX04] Peng Ning, Yun Cui, Douglas S. Reeves, and Dingbang Xu. Techniques and Tools for Analyzing Intrusion Alerts. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):274–318, May 2004.

[Neu] Peter G. Neumann. RISKS Digest: Forum on Risks to the Public in Computers and Related Systems. ACM Committee on Computers and Public Policy.

[Neu78] Peter Neumann. Computer Security Evaluation. In *1978 National Computer Conference, AFIPS Conference Proceedings*, volume 47, pages 1087–1095, 1978.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005.

[OBM06] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A Scalable Approach to Attack Graph Generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 336–345, 2006.

[ON01] William Osser and Alex Noordergraaf. *Auditing in the Solaris Operating Environment*. Sun Microsystems, Inc., February 2001.

[PB07] Sean Peisert and Matt Bishop. How to Design Computer Security Experiments. *submitted to the Fifth World Conference on Information Security Education (WISE)*, February 2007.

[PBKM05] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Principles-Driven Forensic Analysis. In *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, pages 85–93, Lake Arrowhead, CA, October 20–23, 2005.

[PBKM07a] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Analysis of Computer Intrusions Using Sequences of Function Calls. *conditionally accepted by IEEE Transactions on Dependable and Secure Computing (TDSC)*, January 2007.

[PBKM07b] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Toward Models for Forensic Analysis. In *Proceedings of the 2nd International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE) (to appear)*, Seattle, WA, April 10–12 2007.

[Pei05] Sean Peisert. Forensics for System Administrators. *;login:*, 30(4), August 2005.

[PFMA05] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot — a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 2005 USENIX Security Symposium*, 2005.

[PH99] Raju Pandey and Brant Hashii. Providing Fine-Grained Access Control for Java Programs. In *Proceedings of the Thirteenth European Conference on Object Oriented Programming (ECOOP)*, June 1999.

[Pic87] Jeff Picciotto. The Design of an Effective Auditing Subsystem. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 1987.

[Roe99] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Large Installation System Administration Conference (LISA)*, Seattle, WA, November 7–12 1999.

[SB03] Fred Chris Smith and Rebecca Gurley Bace. *A Guide to Forensic Testimony: The Art and Practice of Presenting Testimony As An Expert Technical Witness*. Addison Wesley Professional, 2003.

[Sch99] Bruce Schneier. Attack Trees: Modeling Security Threats. *Dr. Dobb's Journal*, 24(12):21–29, December 1999.

[Sch00] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, February 2000.

[See89a] Donn Seeley. A Tour of the Worm. In *Proceedings of the Winter 1989 Usenix Conference*, San Diego, CA, 1989.

[See89b] Donn Seeley. Password Cracking: A Game of Wits. *Communications of the ACM (CACM)*, 32(6):700–703, June 1989.

[Shi95] Tsutomu Shimomura. Technical details of the attack described by Markoff in NYT. `http://www.takedown.com/coverage/tsu-post.html`, January 25 1995.

[Shi97] Tsutomu Shimomura. Testimony before the United States House of Representatives Committee on Science, Subcommittee on Technology. `http://www.house.gov/science/shimomura_2-11.html`, February 11 1997.

[SHJ$^+$02] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, CA, 2002.

[Sib88] W. Olin Sibert. Auditing in a Distributed System: SunOS MLS Audit Trails. In *Proceedings of the 11th National Computer Security Conference*, pages 82–90, October 1988.

[Sin05] Abe Singer. Tempting Fate. *;login:*, 30(1):27–30, February 2005.

[SK99] Bruce Schneier and John Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, May 1999.

[SL03] Tye Stallard and Karl Levitt. Automated Analysis for Digital Forensic Science: Semantic Integrity Checking. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, December 8-12 2003.

[SM90] Kenneth F. Seiden and Jeffrey P. Melanson. The Auditing Facility for a VMM Security Kernel. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, 1990.

[SM96] Tsutomu Shimomura and John Markoff. *Takedown.* Hyperion Press, 1996.

[Som98] Peter Sommer. Intrusion Detection Systems as Evidence. In *Proceedings of the First International Workshop on Recent Advances in Intrusion Detection (RAID)*, 1998.

[Spa89] Eugene H. Spafford. Crisis and Aftermath. *Communications of the ACM (CACM)*, 32(6):678–687, June 1989.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[Ste00] Peter Stephenson. The Application of Intrusion Detection Systems in a Forensic Environment (extended abstract). In *The Third International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2000.

[Sto88] Cliff Stoll. Stalking the wily hacker. *Communications of the ACM*, 31(5):484–497, May 1988.

[Sto89] Cliff Stoll. *The Cuckoo's Egg.* Pocket Books, 1989.

[Sun89] Sun Microsystems, Inc. NFS: Network File System Protocol Specification. RFC 1094, March 1989.

[SV05] Srianjani Sitaraman and S. Venkatesan. Forensic Analysis of File System Intrusions using Improved Backtracking. In *Proceedings of the Third IEEE International Workshop on Information Assurance*, pages 154–163, 2005.

[SZ97] Richard T. Simon and Mary Ellen Zurko. Adage: An Architecture for Distributed Authorization. Technical report, OSF Research Institute, Cambridge MA, 1997.

[TC90] Henry S. Teng and Kaihu Chen. Adaptive Real-Time Anomaly Detection Using Inductively Generated Sequential Patterns. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, 1990.

[Tho84] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[TL00] Steven J. Templeton and Karl Levitt. A Requires/Provides Model for Computer Attacks. In *Proceedings of the 2000 New Security Paradigms Workshop (NSPW)*, pages 31–38, Ballycotton, County Cork, Ireland, 2000.

[TM02] Kymie M.C. Tan and Roy A. Maxion. "Why 6?" — Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 188–201, Oakland, CA, 2002.

[van94] Leendert van Doorn. nfsbug.c. `ftp://ftp.cs.vu.nl/pub/leendert/nfsbug.shar`, April 1994.

[van98] Leendert van Doorn. nfsshell. `ftp://ftp.cs.vu.nl/pub/leendert/nfsshell.tar.gz`, May 1998.

[Ven92] Wietse Venema. TCP WRAPPER: Network monitoring, access control, and booby traps. In *Proceedings of the 3rd USENIX Security Symposium*, September 1992.

[VO01] Gustavo Villavicencio and J. N. Oliveira. Reverse Program Calculation Supported by Code Slicing. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 35–45, October 2001.

[VVKK04] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. A Comprehensive Approach to Intrusion Detection Alert Correlation. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 1(3):146–169, July–September 2004.

[Wee95] Christopher Wee. LAFS: A Logging and Auditing File System. In *Proceedings of the 11th Annual Computer Security Applications Conference (ACSAC)*, pages 231–240, December 1995.

[Woo01] Avishai Wool. Architecting the Lumeta Firewall Analyzer. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[XBH03] Min Xu, Rastislav Bodik, and Mark D. Hill. A “Flight Data Recorder” for Enabling Full-System Multiprocessor Deterministic Replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 122–133, 2003.

[YMS$^+$06] Lihua Yuan, Jiannina Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[ZHR$^+$07] Jingmin Zhou, Mark Heckman, Brennan Reynolds, Adam Carlson, and Matt Bishop. Modelling Network Intrusion Detection Alerts for Correlation. *ACM Transactions on Information and System Security (TISSEC)*, 10(1), February 2007.

[ZMAB03] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C. Bressoud. Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN)*, 2003.

[ZS96] Mary Ellen Zurko and Richard T. Simon. User-Centered Security. In *Proceedings of the 1996 New Security Paradigms Workshop (NSPW)*, pages 27–33, Lake Arrowhead, CA, September 1996.