Toward Models for Forensic Analysis

Sean Peisert* Matt Bishop† Sidney Karin Keith Marzullo

Abstract

The existing solutions in the field of computer forensics are largely ad hoc. This paper discusses the need for a rigorous model of forensics and outlines qualities that such a model should possess. It presents an overview of a forensic model and an example of how to apply the model to a real-world, multi-stage attack. We show how using the model can result in forensic analysis requiring a much smaller amount of carefully selected, highly useful data than without the model.

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)

Forensic analysis is the process of understanding, re-creating, and analyzing arbitrary events that have previously occurred. It seeks to answer the questions of *how* an intrusion occurred and *what* the attacker did during the intrusion. Forensic analysis also refers to the derivation of information for use in court. That aspect can build on what we describe in this paper, but we focus on the more general notion of collecting data to reconstruct events and activities.

Current computer forensic analysis uses information gathered *ad hoc*, because applications log it for debugging purposes (syslog) or because it "seems" important (BSM [Sun01]). The result is that current forensic systems record too much of the wrong data, making analysis difficult or impossible. Computer forensic analysis would benefit from a rigorous model to help balance the usefulness of logged data against the practicality of performing both logging and auditing.

Practical forensic analysis has traditionally traded off accuracy against the amount of data recorded. A forensic solution at one extreme [PBKM05] is to record everything, including all memory accesses explicitly made in an intruder's program, or anything that

*S. Peisert, S. Karin, and K. Marzullo ({peisert,karin,marzullo}@cs.ucsd.edu) are with the Department of Computer Science & Engineering at the University of California, San Diego. This research was supported in part by award ANI-0330634, "Integrative Testing of Grid Software and Grid Environments," from the National Science Foundation, and by the AFOSR MURI Grant 41131-6865.

†M. Bishop (bishop@cs.ucdavis.edu) is with the Department of Computer Science at the University of California, Davis. He was supported in part by awards CCR-0311671 and CCR-0311723 from the National Science Foundation to the University of California, Davis.

transitions a system from one state to another [Gro97]. The other end of the spectrum is to record high-level data such as syslog messages or data focused in one particular area. Examples include filesystem data from *Tripwire* [KS94] and *The Coroner's Toolkit* [FV], or connection data from *TCP Wrappers* [Ven92]. Attempts [PBKM06] to find a middle ground still try to solve forensic problems with *available* data, rather than seeking alternative ways to obtain *necessary* data.

An alternative is to determine what data is necessary to understand past events. This results in a smaller amount of carefully selected, highly useful data. Whereas recording everything, might give maximum *utility*, it is impractical, if not infeasible. Similarly, our earlier ad hoc approach of recording all function calls [PBKM06] also gives high utility, but at substantial cost.

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 previously identified five, high-level principles of forensic analysis [PBKM05]. 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:

Principle 1: *Consider the entire system.*

Principle 2: *Log information without regard to about expected failures and attacks*

Principle 3: *Consider the effects of events, not just the actions that caused them.*

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

Principle 5: *Present events in a way that can be analyzed and understood by a human forensic analyst.*

We present a model of critical elements of forensic analysis, from which we derive what needs to be recorded to perform effective forensic analysis while considering cost in performance and other measures. We discuss a method of capturing more *useful* data rather than *more* data, and as a result, reduce the burden on both the computer system and the human analyst.

Throughout our work, we assume that our forensic software obtains accurate information from the system, and is able to report that information to the analyst correctly.

For clarity, we define several terms. An *event* is some action that a user (legitimate or illegitimate) can take themselves, or can automate a computer to take. This does not include hardware manipulation or events that occur within a virtual machine (VM). An *attack* is a sequence of events that violates a *security policy* of the site, whether initiated externally (a break-in) or internally (the "insider problem"). We currently ignore motive in our work. The *goal* of the attack is to achieve a particular violation. An *attack graph* represents an attack as a sequence of goals linked together in a directed graph. Goals that follow other goals depend on the results or effects of achieving the earlier goals.

Section 2 presents related work. Section 3 presents qualities for a good forensic model. Section 4 discusses the premise of our forensic model and section 5 discusses how well our forensic model adheres to the desired qualities. Section 6 applies our model to a worm. Section 7 discusses future work and section 8 concludes the paper.

2: Background and Related Work

As previously mentioned, much of current forensic analysis is ad hoc. The more successful work has led to useful tools, often unified using a "toolbox" approach [FV04]. Unfortunately, the lack of a rigorous model of forensic analysis often causes information to be missed, or captures superfluous information that inflates the amount of storage needed and detracts from the forensic analysis.

For example, 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. Further, identifying the starting point was not a goal of BackTracker or its successors [SV05, KMLC05]. Nor does BackTracker always identify what happens *within* the process, because BackTracker is primarily aimed at a process-level granularity.

Forensix [GFM+05] also records system calls, but 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 has constraints similar to BackTracker. A forensic analyst, for example, has to independently determine which files might have been written to by an intruder's code.

The papers proposing using intrusion detection data as forensic evidence focus on legal admissibility [Som98]. They emphasize that data is already collected in real-time [Ste00], and not the utility of that data.

A few approaches have used forensic models. For example, Gross [Gro97] exhaustively researched the usable data and analysis techniques from unaugmented systems. His goal was to formalize existing auditing techniques. However, Gross's work does not discuss what information is necessary to understand attacks.

Previous research in modeling systems examines the limits of auditing in general [Bis89] and auditing for policy enforcement [Sch00]. 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). None of this work focuses on the specific information needed for forensic analysis.

3: Qualities for a Forensic Model

In addition to adhering to the above 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 the analyst 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 in some cases, modeling *all* possible attacks that enable an attacker to gain certain capabilities on a system is impractical. 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 help an analyst also understand the effects of events as well as the actions that caused them. Hence, a good forensic model should consider the conditions of the system both 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 a forensic analyst 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.

4: Our Approach

In this section, we describe a model that possesses the above qualities. Our model builds upon recent work in detecting multi-stage attacks [TL00, ZHR+07] to develop formalisms that allow us to derive rigorously what forensic data requires logging.

The *requires/provides model* [TL00] considers an attack as consisting of a series of *goals*.

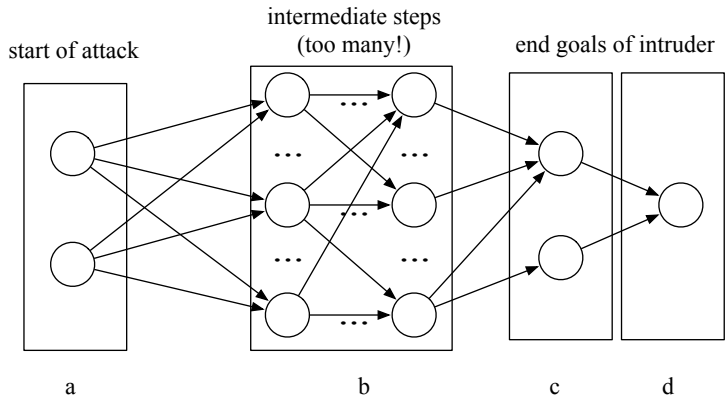

Figure 1. Diagram of a generic attack where circles represent actions. An attack model almost always consists of at least the endpoint (d), but may also include the beginnings (a) and possibly other states near the end (c).

Each *step* in the attack achieves an *intermediate goal*, advancing the attacker closer to the *ultimate goal*. 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 a 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 an attack as a digraph, the intermediate goals being intermediate nodes, the ultimate goal being the destination (sink), the starting points being sources, and the edges being steps in the attack.[1] By *inverting* 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 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 1a) or end of an attack (Figure 1c/d), as opposed to the explosion of possible methods that could be used in the middle of an attack (Figure 1b). Indeed, an intermediate goal could even be part of several attack graphs.

The methodology for using the requires/provides model to perform *goal-oriented attack modeling* 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 for each attack.
3. Specify the pre-conditions and post-conditions ("capability sets") of each goal, outlining the set of data to be logged.
4. 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.

[1] This is a generalization of attack trees [Sch99] used for intrusion detection.

4.1: Choosing Intruder Goals to Model

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

Many intrusion detection research efforts address multi-step attack modeling by correlating alerts from intrusion detection systems, firewalls, and other sources, reducing the amount of information that an analyst would receive, and making the information that they do receive more intelligible. By correlating alerts, a related series of events would appear together, and an analyst would be able to follow their logical progress to success or failure, and to help assess danger or malice from those events. Additionally, redundant alerts could be filtered out. The models for these efforts was done in a number of different ways, and included hosts, networks, or both. All have relied fundamentally on alerts from security software.

But our objective is to do forensic analysis, not intrusion detection. Forensic analysis requires 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. Our model shows what information is *necessary* to analyze an attack; it does not assume that the information recorded to answer the *if* question is sufficient. So, goals are based on specific conditions that lead to logging appropriate data. This approach suggests the controls that need to be implemented to reconstruct the attacks, 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 [Bis03]. These methods use the axioms of well-defined policies to dictate logging requirements. However, real-world policies mix implicit elements with explicit elements, and are inherently ambiguous. Automatically *discovered security policies* [BP06] or well-defined vulnerability classifications [Bis99] may eliminate some ambiguity.

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 paper, 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.

4.2: 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 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 the attacker used. Given the graph, the analyst can ask what changes in the state of the system each attack step introduces. This provides the information needed to analyze the change. This ties forensics to a model of attack, and provides a basis for collecting specific information.

Capturing precise exploits used to achieve intermediate results may rely upon recording events 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, an exploit might be inferred with a high degree of confidence.

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.

"Capabilities" are not the same as "implications." A *capability* is something that is immediately gained through a step in a process, such as knowing a password. An *implication* is a theoretical possibility that hypothesizes certain information, such as the ability to log in to a system using the password. The model addresses the former, not the latter.

Each goal is modeled by a *capability pair* (a 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.

A capability has six components: source, destination, credential, action, service, and property. *Source* and *destination* refer to addresses. Our examples all contain IP addresses. A *credential* is a level of access. Generally, an access level is represented by a uid or gid for cases involving local access. In most cases, it refers to a uid of *root*, a (specific) *user*, *no* user, or *ANY* user. An *action* is an event such as read, write, or connect. The action may be at a high layer of abstraction (UNIX shell command), an intermediate level (library or function call), or low level (machine instruction, keystroke). A *service* is a level of functionality, such as a program, the filesystem, or a portion of the kernel. A *property* is the portion of a service or object (e.g. permissions, running status, version, path, or owner) under consideration. For example, a user might *exec* (action) the *code* (property) of a *shell* (service), and might also *modify* (action) the *permissions* (property) of that same *shell* (service).

Previous work in intrusion detection [ZHR+07] uses this 6-tuple 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.

4.3: Extracting and Interpreting Logged Data

Data can be logged at different levels of abstraction. System-level logging typically represents an action as a sequence of system calls, for example mapping a file name into an address, locating the file with that address, using memory mapping to map the file into memory, and returning a descriptor to the caller. Application-level logging typically represents an action as a high-level event, such as a file being opened. Thus, the low-level implementation of a high-level abstraction will indicate what to log.

The mapping between system-level logs and application-level logs is critical because of the end-to-end principle.[2] This principle states that logging should be done as close to the

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

source as possible. For systems, the logging should be either at the point of resource access (controlled by the reference validation mechanism or its equivalent, typically the kernel) or at the point of resource use (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; in fact, syslog, a widely used logging mechanism on UNIX-style systems, is susceptible to exactly this attack. 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.

Conversely, a reversal of the translation process enables the aggregation of low-level system call log entries into higher-level abstractions, thereby aiding the process of reconstructing intermediate goals. Zhou et al [ZHR+07] did this to network intrusion detection logs.

Given a requires/provides model of a goal or a set of goals, there are two questions: how do we know what information to log? And, once we have logged that information, does it describe a distinct class of attacks, or even a specific attack?

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. The basic algorithm is this: given an attack graph consisting of goals modeled as requires/provides capability pairs, we walk through the graph from back to front, applying an algorithm that takes the capability pairs as inputs and outputs the logging requirements. In cases where a particular intermediate goal is unknown, we can set upper and lower bounds based on the capabilities of the goals that precede and follow it. In cases where multiple goals are required before proceeding to the next, we take the union of the capabilities required. The translation process for individual capability pairs is discussed briefly in section 7.

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 in determining the point to place the logging mechanism.

5: Evaluating Our Model

In this section, we compare our model to the set of qualities described in section 3:

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. Section 7 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 the 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*: Section 7 describes a method that will add this quality to our model.

Our forensic model meets most of the qualities necessary to analyze attacks well.

6: Example: The 1988 Internet Worm

The 1988 Internet Worm [ER89], 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.

The ultimate goal of the worm was to spread. The worm took the following steps, which are characteristic of a broad class of worms:

1. Run multiple exploits against a system.
2. Once on the system, invoke a shell running as the user who owned the process that was exploited, or as a user whose account was compromised (either by guessing a password or through trust relationships).
3. Spawn a copy of itself approximately every 3 minutes to refresh its appearance of use.
4. Meanwhile, try to spread to other machines.

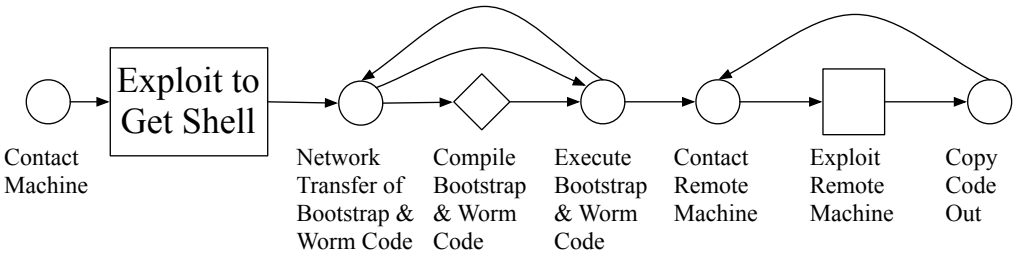

Figure 2. The attack graph used by the 1988 Internet Worm

Figure 2 shows a representation of this attack.[3] The goals represented in the figure as circles are known. The diamond represents a known goal that is "optional" and may not occur. The squares and rectangles represent unknown sequences of goals; they may contain too many possible intermediate goals to enumerate. Therefore, we describe their upper and lower bounds, derived from the pre-conditions and post-conditions of the goals that precede and follow them. The loops in figure 2 represent the goals that may by repeatedly achieved—transferring code to the target more than once, or exploiting multiple remote machines.

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 requires/provides capability pair for the Contact goal looks as follows:

- Requires: A process running on the target machine that is listening on an open network port.
- Provides: Access to target system; details uncertain.

The Transfer goal is slightly more complicated. To transfer a readable file to the target machine, the attacker need to be a user on the local machine, needs to be a user on the target machine, and needs to 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:

- Requires: A user on the local machine and a user on the target machine who has a process capable of requesting and receiving network data.
- Provides: A file (the worm) that the user on the target system is capable of reading.

The bounds on the exploit to get a shell can be determined from the *provides* capabilities of the Contact goal and the *requires* capabilities of the Transfer goal. Somewhere between the two, the ability to transfer a file, readable by a particular user on the target machine, must be acquired. Therefore, the bounds on the exploit involve the specific set of IP addresses used to exploit the target machine, acquire access, and provide the ability to execute programs capable of requesting or receiving a file.

The Compile goal is optional, but ensures that a program can be executed:

- Requires: Local user that can read the worm code, execute the compiler, and write a binary.
- Provides: Local user that can execute the binary.

The Execute goal starts the program:

- Requires: Local user that can execute the binary.
- Provides: A running binary (the worm code) owned by a local user.

The ContactOut goal starts the propagation of the worm:

- Requires: Local user who owns a running program.
- Provides: Ability to communicate with other machines on the network.

The TransferOut goal involves transferring the worm code to the remote target:

[3]The Internet worm could be modeled at a higher or lower level of abstraction. This level is appropriate for our analysis.

- Requires: Shell owned by a local user on the remote target.
- Provides: Nothing (the code was transferred).

The ExploitRemote goal is unknown and is formed by using bounds from the previous and succeeding goals:

- Requires: Ability to communicate with other machines on the network.
- Provides: Shell owned by a local user on the remote target.

We have developed a model that describes the algorithm for extracting the logging information from these requires/provides capability pairs, though the details of the algorithm or an example of walking through it in detail is beyond the scope of this paper. However, the summary of the results of applying such an algorithm is simply a small subset of system calls with a small amount of contextual information relating to user IDs, file paths, and network addresses.

The example that we have shown in this section suggests that an implementation of this should result in forensic data of the efficiency and effectiveness that a forensic analyst would desire, as opposed to current solutions. The results of carrying the application of this forensic model to its conclusion are not yet known, but in the next section, we outline the process that we will use to obtain those results.

7: Future Work

Many open questions remain. A key component of validating this model experimentally is to establish a relationship between intruder actions and logged events. As we said earlier, this "translation function" is system-dependent. Ideally, we would like to show that a given set of logged information corresponds to exactly one exploit (or a small set of exploits). Further, the intruder goals should be based upon the policy being violated. Combining a well-defined policy language and the ability to translate between high-level policies and goals and low-level, system-dependent implementation details by using policy discovery [BP06] would ease this task. Another open question is how to augment existing logging systems so that they are robust enough to use with our model.

If a forensic analyst finds evidence of a midpoint of a multistage attack, she 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, 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. This reduces the amount of data that must be analyzed—a serious problem in current 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 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.

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.

Forensics involving concurrent sessions requires information about the (relative) order of events, which this work does not address. This area requires more work.

A *unique path identifier (UPI)* simplifies associating seemingly different segments of attack trees. A UPI is a tuple that indicates an attacker's position in a tree, associates it with known prior positions, and uniquely identifies all known distinct traversals of a tree. Unfortunately, the markers that 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. On certain UNIX-like systems, the "audit user ID" (AUID) will not 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 being logged in from (because this is difficult to change), the process identifier (PID), provided child processes maintain a tie to it, and a unique counter to disambiguate multiple occurrences of the same AUID, IP address, and PID.

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 paper, the data captured from the known goals helps to compensate for the absence of a mechanism to enforce logging a strict relative ordering of events.

8: Conclusions

We have presented a set of qualities that we believe that a good forensic model should possess, as well as an overview of our own forensic model that does possess those qualities. We do not believe that the qualities that we have presented for a forensic model are either complete or rigid, but we believe that a model, such as our own, that we have derived from them, should be both more effective and more efficient at logging and auditing.

Our model is a piece in the puzzle that describes a relationship between data needed to analyze and understand events, and the events themselves. We have, therefore, also presented a series of steps outlining future work that we believe will help to determine whether the forensic model is ultimately more efficient and effective than existing solutions.

While current operating systems lack well-structured forensic capabilities, new systems should be capable of fostering effective forensic analysis at minimum cost, and existing systems could be augmented to add those capabilities. We believe that a system using a model based on the qualities that we presented and used for our own forensic model, should be much more successful at both efficiently and effectively performing forensic analysis.

References

[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.

[Bis99] Matt Bishop. Vulnerabilities Analysis. In *Proceedings of the Second International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 125–136, September 1999.

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, Boston, MA, 2003.

[BP06] Matt Bishop and Sean Peisert. Your Security Policy is *What*?? Technical Report CSE-2006-20, University of California at Davis, 2006.

[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.

[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.

[Gro97] Andrew H. Gross. *Analyzing Computer Intrusions*. PhD thesis, University of California, San Diego, Department of Electrical and Computer Engineering, 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.

[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.

[PBKM06] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Analysis of Computer Intrusions Using Sequences of Function Calls. *Submitted to IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2006.

[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.

[Som98] Peter Sommer. Intrusion Detection Systems as Evidence. In *Proceedings of the First International Workshop on Recent Advances in Intrusion Detection (RAID)*, 1998.

[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.

[Sun01] Sun Microsystems, Inc. *Auditing in the Solaris™ Operating Environment*, February 2001.

[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.

[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.

[Ven92] Wietse Venema. TCP WRAPPER: Network monitoring, access control, and booby traps. In *Proceedings of the 3rd USENIX Security Symposium*, September 1992.

[ZHR+07] Jingmin Zhou, Mark Heckman, Brennan Reynolds, Adam Carlson, and Matt Bishop. Modelling Network Intrusion Detection Alerts for Correlation. *To Appear in ACM Transactions on Information and System Security (TISSEC)*, 2007.