# DETERMINING INTRUSION ACTIVITY FOR FILE-SYSTEM RECOVERY

by

Kai Yi Po

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

# Abstract

Determining Intrusion Activity for File-System Recovery

Kai Yi Po

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2005

Recovery from intrusions is typically a very time-consuming and error-prone task because the precise details of an attack may not be known. The wide availability of attack toolkits that install modified utility programs and erase log files to hide an attack further complicates this problem. This thesis explores a fast and accurate method for determining intrusion activity for file-system recovery. Given an audit log of all system activities, our approach uses dependency analysis to determine the set of intrusion-related activities. This approach effectively detects all attack-related activities, but it can falsely mark legitimate activities as related to an intrusion. Hence, we propose various enhancements to improve the accuracy of the analysis. This approach is implemented as part of the Taser intrusion recovery system. Our evaluation shows that Taser is effective in recovering from the damage caused by a wide range of intrusions and system management errors.

# Acknowledgements

First and foremost, I would like to thank Ashvin Goel for his thoughtful supervision, his financial support and the large amount of time we spent discussing challenging issues in computer security.

I am thankful to Kamran Farhadi and Zheng Li, with whom I worked on many aspects of the Taser project.

I would also like to thank Eyal de Lara for his time and his valuable input.

Finally, I would like to thank the University of Toronto as well as the Department of Electrical and Computer Engineering for their financial support.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Computer intrusions have become common today because of the widespread availability of automated attack tools. According to CERT [5], the number of intrusion incidents has doubled approximately each year since 1997. When these incidents occur, the attacked systems need to be recovered. The recovery of file-system data is one of the most error-prone and time-consuming tasks. This typically involves many steps: installation of a new system image that includes the operating system and all applications, installation of software patches that fix known vulnerabilities, and retrieval of uncorrupted user data. Each of these recovery steps is manual, tedious and time-intensive.

Today, snapshot-based file-systems [23, 28] provide a well understood and commonly deployed recovery solution [32]. This method gets rid of all corrupted data, but unfortunately, it also gets rid of useful data not related to the intrusion, which has to be manually retrieved or recovered separately.

Given the high human costs associated with recovery, we argue that sacrificing some machine and networking resources for automating the recovery process should be an attractive proposition. Moreover, with the rapid and continuous decline in computing, networking, and storage costs, logging *all* system operations, which is needed for recovery, is now technically and economically feasible [29, 7, 16, 11].

This thesis focuses on analyzing system operations to determine the set of operations that are related to an intrusion. These operations are called *tainted* operations and the analysis process is called *tainting* analysis. Once the set of tainted operations is determined, file-system data can be *selectively* recovered by *reverting* the effects of these tainted operations. This approach has similarities with system software upgrade where the upgrade can be selectively rolled back without affecting the rest of the system [3, 13].

Our tainting approach is implemented as part of the Taser intrusion recovery system. Taser consists of three major components: auditor, analyzer, and resolver. Taser uses the Forensix system [11] as the auditor to provide an accurate audit log of file, process and socket related system-call operations. The Taser analyzer, which is the focus of this thesis, uses the data provided by the auditor to correlate objects based on causal dependencies. Objects are marked as tainted if they depend on an attack-related object. Finally, the Taser resolver takes the set of tainted objects and generates recovery actions for these objects so that the effects of all tainted operations are reverted.

## 1.1 Research Approach

The goal of the Taser analyzer is to identify the set of tainted operations using causal dependencies between kernel-level objects such as processes and files. The analyzer creates dependencies between these objects based on the flow of information that occurs during system-level operations. A causal dependency relationship between two objects occurs when there is an indirect flow of information across them via other objects. Starting from some known tainted objects, the analyzer marks all causally dependent objects as tainted.

Unfortunately, sometimes legitimate user operations may unknowingly interact with attack-related objects. For example, during a daily examination of the various log files in the system, the administrator reads a file that is mangled by an attacker. Such interaction establishes a false dependency relationship and renders the legitimate read operation tainted.

To avoid creating such false dependencies, we enhance the analyzer to ignore certain dependencies during analysis. The choice of dependencies involves an inherent trade-off. These dependencies can be chosen conservatively. For example, any interaction between the kernel objects may be used to taint an object. This approach taints all attack-related operations, but it may also mistakenly mark legitimate operations as tainted, which causes the resolver to issue recovery actions to revert these legitimate operations. In contrast, when some of the dependencies are ignored, the number of mistakenly tainted operations may be reduced, but this approach can miss some attack-related operations. The analyzer exposes this trade-off by providing a choice of tainting policies from conservative to optimistic. The more optimistic policies ignore more dependencies. For example, an optimistic policy may assume that reading certain log files does not cause a process to be tainted.

The analyzer consists of two main phases: trace and propagation. The trace phase attempts to find the attack source objects such as a socket or a server process from which an attack originated. This phase starts with an externally provided or known tainted object such as a file created by an attacker and then traverses causal dependencies in reverse order to find potential attack sources. The attack source objects are fed into the propagation phase, which taints all the objects causally dependent on the attack sources.

The propagation phase provides a number of tainting policies that successively ignore certain dependencies. We find that the differences in the outputs of these policies aid in determining the set of attack-related objects and operations. When the propagation phase is run repeatedly with different policies, it reveals a common set of tainted objects that are likely to be attack-related, while the differences in the outputs of these runs are the ambiguous objects and whether they are attack related needs to be manually determined. Our evaluation shows that the number of ambiguous objects is typically small compared to the total number of file-system objects that have been modified since an attack and hence the precise set of attack-related objects and operations can be determined quickly.

## 1.2 Contributions

The analyzer provides a fast and accurate methodology for determining intrusion activity for file-system recovery. It automates the process of identifying damage caused by an intrusion using a dependency-based tainting analysis method. To achieve accuracy, it provides flexible dependency policies. The analyzer is fully implemented as part of the Taser intrusion recovery system. A detailed evaluation in this thesis shows that the analyzer correctly taints file-system objects for a wide range of intrusions, as well as erroneous system management activities. The analyzer is used in the context of Taser and as such, we also provide a performance evaluation of Taser.

## 1.3 Thesis Structure

The rest of the thesis provides the details of the analyzer. Chapter 2 provides an overview of the other components of Taser, namely the auditor and the resolver. Chapter 3 presents the basic design of the analyzer that can perform a conservative analysis. Chapter 4 describes enhancements that improve the accuracy of the analysis. Chapter 5 discusses the implementation of the analyzer in detail. Chapter 6 provides an evaluation of the analyzer in terms of accuracy and performance. Chapter 7 discusses the related work in the area. Finally, Chapter 8 presents the conclusions and directions for future work.

# Chapter 2

# An Overview of the Taser Recovery System

The Taser system consists of three components: the auditor, the analyzer and the resolver. This chapter provides an overview of the auditor and the resolver components of Taser. The focus of this thesis is the analyzer, and it is discussed in detail in the later chapters.

## 2.1 The Taser Auditor

The Taser auditor consists of the Forensix system, which provides an audit log of all system calls performed in a target system. In particular, Forensix monitors and logs all the system calls and their arguments and securely stores this audit log into a database at a backend system.

Figure 2.1 shows the Forensix architecture. The target system, which hosts publicly-available services, is potentially vulnerable. The Forensix logger monitors the system calls from the target system's kernel to retrieve the relevant audit information. This audit log is transmitted to the backend system via a private network and saved into append-only files at the backend system. Then the data is batch-loaded into a database periodically. The separation of the audit log from the target system ensures that the audit data cannot be easily destroyed.

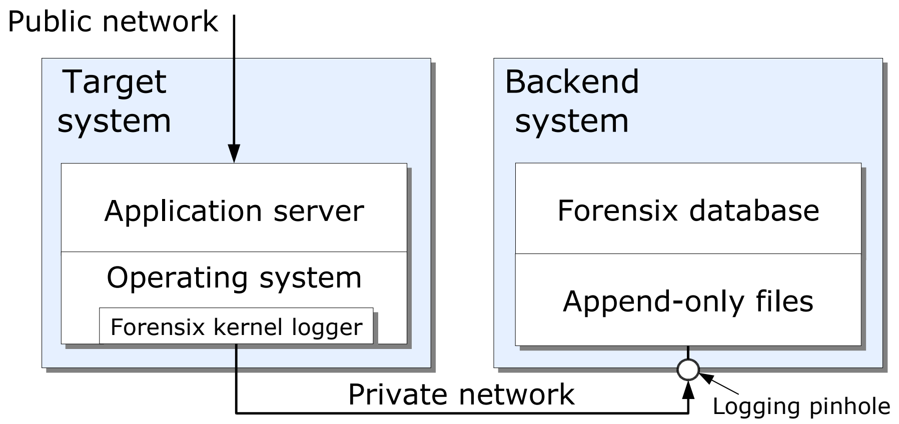


Figure 2.1: The Forensix architecture

The Forensix audit system has three characteristics that make it suitable for kernel-level auditing: resistance to intrusions, unambiguous system-call attribution, and race-free auditing.

**Resistance to Intrusions** The auditing system must be resistant to attacks. Since the Forensix logger runs in the kernel on the target system, the logger is as vulnerable as the target system's kernel. The statistics we collect from Secunia [25] show that for the three distributions of Redhat Linux (Fedora Core 1, 2 and 3), the breakdown of the number of known kernel to application-level vulnerabilities is 9/74, 9/132 and 6/112. In other words, kernel vulnerabilities accounted for 10% or less of all vulnerabilities. While security statistics are never conclusive [18], we believe that securing the kernel code is easier than securing all applications. To reduce the risk of kernel intrusions, Forensix uses LIDS [34] to disable 1) user-level writes to kernel memory, 2) user-level writes via the raw disk interface, 3) writing to the kernel or Forensix binary files, and 4) the loading of kernel modules. These simple measures make current root-kits ineffective against the kernel [10, 24], and hence against the Forensix logger as well.

**Unambiguous System-Call Attribution** The Forensix kernel logger has direct access to system-call arguments because it wraps system calls to get audit data. The system-call arguments by themselves do not unambiguously determine what kernel objects are involved in the call. For example, the file object accessed when `./passwd` is opened depends on the current working directory of the calling process. To unambiguously identify the kernel objects involved in system calls, the logger retrieves the kernel-level identifier of each object, which is unique at any given point in time. However, these identifiers may be re-issued by the kernel for storage efficiency. Since Forensix needs to identify objects uniquely over time, it attaches an extra field, which is usually a timestamp, to distinguish kernel objects over time.

**Race-Free Auditing** The Forensix logger is implemented as wrappers to system calls. These wrappers are prone to concurrency races. For example, a potential race condition occurs when a `open` system call is issued concurrently with a `rename` system call. Between the time when Forensix audits the `open` system call's file name and the file is opened, if the `rename` system call renames another file to the file name being opened, then Forensix would record the wrong file object for the `open` system call. To avoid these races [9], Forensix uses the Linux security modules facility [33] to gather further information that helps to precisely determine the identity of kernel objects when they are accessed during system calls. In the example above, the file object ID (the inode on UNIX file systems) is recorded during the open call.

## 2.2 The Taser Analyzer

The Taser analyzer is described in detail in later chapters. Briefly, it traverses the audit log provided by the auditor and taints objects that are causally dependent on a seed object. The seed object is provided externally and is known to be related to an attack.

## 2.3 The Taser Resolver

The goal of the resolver is to revert tainted file-system operations but preserve legitimate operations. The resolver's architecture is shown in Figure 2.2. It takes as input a file-system snapshot, the set of tainted file-system objects and operations generated by the analyzer, and

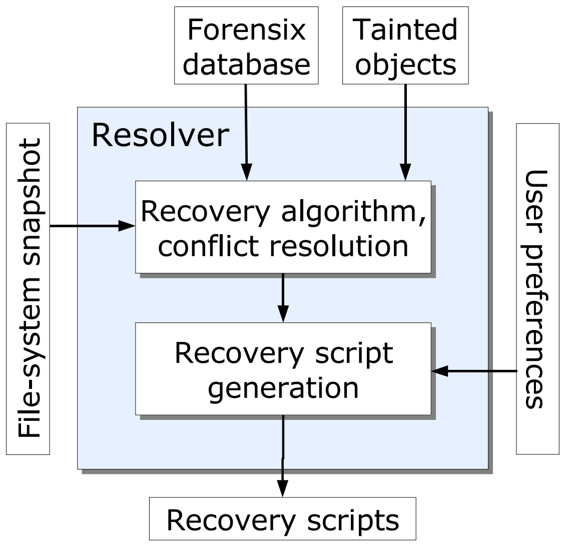


Figure 2.2: The resolver architecture

the audit log created by the auditor. To revert operations, the resolver uses a *selective redo* algorithm that only replays the legitimate operations in the audit log. The resolver assumes that recovery starts with an immutable file system so that the file-system state does not change during recovery.

The selective redo algorithm makes two optimizations to improve the performance of the recovery process. First, the recovery is started using the current state of the file system. At this state, all non-tainted objects are at their correct state and so no recovery action is needed for these objects. To recover a tainted object, the resolver obtains an initial version of the object from the file-system snapshot and sequentially replays the object's legitimate modification operations since the snapshot. Second, the algorithm recovers file name, content, and attribute operations separately, which helps in optimizing name and attribute recovery. At each operation that modifies name or attribute, the auditor captures the complete resulting state of the object. As a result, only the last legitimate operation needs to be replayed to recover a tainted name or attribute. This optimization is not applicable for content recovery because the auditor only captures changes in state but not the entire state for content operations.

While the analyzer enhancements described later in Chapter 4 help achieve accuracy during recovery, they may introduce conflicts when legitimate operations depend on tainted operations. In such cases, reverting the tainted operations may result in cascaded reverting of the dependent legitimate operations which violates the resolver's goal of preserving all legitimate operations. To handle these conflicts, the resolver defines various conflict resolution methods. These resolution methods either reorder operations or isolate the tainted objects by recovering them with special name extensions. For example, a legitimate operation creates a file under a directory created by an intrusion. The resolver finds a conflict: it cannot remove the tainted directory due to the legitimate file in it. The resolver handles this case by renaming the tainted directory with a `.nonexistent` extension as an alternative way to remove the tainted directory while preserving the legitimate file. The detailed information regarding the conflict resolution methods is described by Goel et al. [12].

# Chapter 3

# The Basic Analyzer

The goal of the analyzer is to determine the set of tainted file-system objects. To do so, the interactions between three types of kernel objects, sockets, processes and files, are analyzed to determine causal dependencies. Socket connections form initiating points for remote attacks, processes issue operations that create other dependent processes or files, and file accesses cause additional dependencies, and, in addition, files are the persistent state of the system that need to be recovered.

A dependency occurs when information flows from one object to another via an operation. When an object is tainted, the taint propagates along the direction of the dependency. In other words, a tainted source object of a dependency renders the sink object tainted. A dependency is denoted by $O_{src} \xrightarrow{op} O_{snk}$ where $O_{src}$ is a source object, $O_{snk}$ is the sink object, and $op$ is the relevant operation. For example, when a process writes to the file, the file becomes dependent on the process. Similarly, a process becomes dependent on a file when it reads the file.

## 3.1 Dependency Model

Table 3.1 shows the operations between the objects that are considered by the analysis. A process to process dependency occurs when a child process is forked, which captures a tainted process hierarchy, and when IPC and signal-based communication occurs between processes.

| **Dependencies** | **Operations** |
|---|---|
| Process → Process | IPC, Signals |
| Process → Process | Fork |
| Process → File Content | Write/remove file content |
| Process → File Name | Write/remove file name |
| Process → File Attribute | Write/remove file attribute |
| Process → Socket | Write |
| File Content → Process | Execute |
| File Content → Process | Read file content |
| File Name → Process | Read file name |
| File Attribute → Process | Read file attribute |
| Socket → Process | Read |

Table 3.1: Dependencies between processes, files and sockets

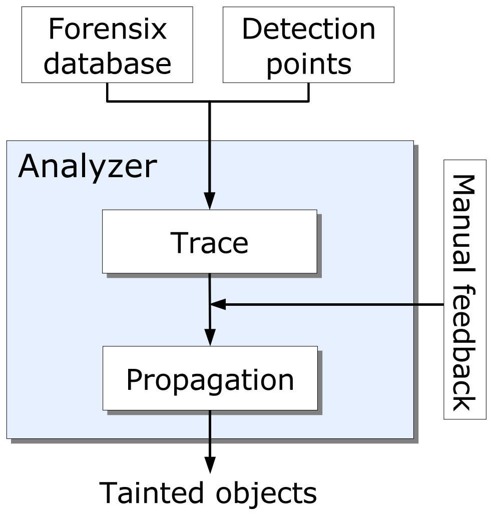


Figure 3.1: The analyzer architecture

A process to file dependency occurs when a process writes to the content, name or attribute (permissions and ownership) of a file, or when a process removes the content, name, or attribute of a file. Note that file content, name and attribute are treated as *separate* objects because this separation allows finer-grained analysis. A file to process dependency occurs either when a process executes a file or reads the content, name or attribute of the file. For example, suppose a directory's attribute is tainted. A process accessing that directory will then become tainted. Each component of a path name is considered as a separate object, and hence, when a process accesses a path name in a system call, a dependency occurs from every component of the path name to the process.

A process to socket dependency occurs when a process writes to a socket, and a socket to process dependency occurs when a process reads from a socket. Given that attacks often originate from the network, the taint propagation typically starts from socket objects.

## 3.2 Analysis Phases

The analyzer derives the set of tainted objects using the dependencies defined above. Figure 3.1 shows the architecture of the analyzer. The analysis starts with an initial set of tainted objects that are assumed to have been detected by an IDS or by a administrator. These tainted objects are known as the detection points. These objects could either be the source of an attack (such as a NIDS alert about a connection that transfers malicious content) or the result of an attack (such as a HIDS alerts about a strange file). The analysis proceeds in two phases, a trace phase and a propagation phase, which we describe next.

### 3.2.1 Trace Phase

When the detection points are not the source of an attack, it is necessary to trace back to the sources to determine all objects that are related to the attack. To trace the sources, the trace phase is given the detection points as tainted objects and a conservative estimate of the attack time interval. The trace phase traverses all operations within the given time interval in reverse time order so that the taint propagates from sink objects to source objects in this phase.

The set of tainted objects resulting from the reverse tainting analysis helps the administrator choose objects that are deemed to be an attack source. For example, the administrator may run the trace phase using a single suspicious login session as the detection point to find all other sessions from the same attacking IP address. To further aid the analysis, the output is grouped according to various criteria (such as number of dependent objects, processes that have accepted remote connections, setuid processes, remote sockets, setuid files, etc.)

Since it is unlikely that all potential attack sources are selected as the attack source for the propagation phase, the objects tainted during the trace phase may not be tainted in the propagation phase. So, all objects are marked untainted at this point, except those that are selected as the sources of the attack.

### 3.2.2 Propagation Phase

The propagation phase starts with the attack-source objects determined above. This phase traverses all the operations from the attack time in time order and propagates the taint to all causally dependent objects. In the basic model, once an object is marked tainted, it typically remains for the lifetime of the object. However, for file names, it lasts forever. Hence, when a tainted file name is removed from the system and is later recreated, the new file name object still remains tainted. The output of this phase is the set of tainted objects, ready to be used by a recovery tool.

The basic dependency model presented above captures every object that is potentially related to the source of an attack. However, sometimes a legitimate objects may unknowingly access a tainted objects, which creates a false dependency and the legitimate objects is then falsely tainted. The next chapter will describe enhancements to the analyzer that improve the accuracy of the analysis by ignoring these false dependencies.

# Chapter 4

# The Enhanced Analyzer

The dependency model presented above taints attack-related objects but may taint legitimate objects also. While all operations presented in Section 3.1 cause information flow, it is unclear, without detailed program analysis [22], whether the dependencies that are created are critical. For example, a `SIGCHLD` signal sent from a tainted process to its parent process may occur as part of normal activity, and hence the parent process should not be marked tainted solely because of such a signal. Similarly, applications may read and write from `/dev/null` but this file does not cause any explicit information flow.

A solution to the problem described above is to relax the dependencies show in Table 3.1. We define dependency rules as a mechanism to govern the behaviour of the analyzer. The use of these rules, on one hand, reduces the possibility of tainting legitimate objects; on the other hand, it increases the chance that an attacker's operations are left untainted. This trade-off is discussed in the evaluation. Dependency rules control two types of dependency: inter-object dependency (across objects) or intra-object dependency (within an object over time). These are discussed is the following sections.

## 4.1 Inter-object Dependencies

The dependencies that are described in the previous chapter are inter-object dependencies. These dependencies relate two objects based on an operation. Below, we discuss two types of dependency rules: tainting policies and white list, to reduce the number of falsely tainted objects. Tainting policies ignore dependencies caused by certain operations, while white list ignores dependencies caused by certain objects.

### 4.1.1 Tainting Policies

We define six tainting polices that progressively ignore an increasing number of dependencies. These policies range from a conservative policy that creates dependencies by taking into account all operations, to an optimal policy that considers a minimal set of operations.

Table 4.1 shows these policies. The Snapshot policy taints all operations that occur after an attack regardless of any dependencies. It is used for comparison against the other dependency-based policies. The Conservative policy takes all dependencies into account. From the NoI policy onwards, each policy progressively ignores the following operations: IPC/signals, file attribute read, file name read, and file content read. The NoI policy ignores IPC and signal operations, the NoIA ignores IPC, signal, and file attribute read operations, and so on. The NoIANC policy is the most optimistic policy. It includes a minimal set of operations that are considered essential for creating tainting dependencies. These operations include fork that creates the tainted process hierarchy, file or socket writes by a tainted process, execution of a tainted file, and reads from a tainted socket.

### 4.1.2 White List

Besides pruning dependencies based on operations, dependencies can be pruned based on objects as well. When a tainted object is white-listed, its taint does not propagate to the sink object in an operation. A common situation where an object is white-listed is when a shared

| Policy | Description |
|---|---|
| 0 Snapshot | All operations tainted |
| 1 Conservative | All operations shown in Table 3.1 |
| 2 NoI | Ignores IPC, signals |
| 3 NoIA | Further ignores reading file attribute |
| 4 NoIAN | Further ignores reading file names |
| 5 NoIANC | Further ignores reading file contents |

Table 4.1: Tainting policies and operations

file accessed by many processes. If this shared file becomes tainted, all processes that access it will become dependent on it, potentially creating many falsely tainted process objects.

In practice, white list works well in conjunction with the NoI, NoIA and the NoIAN tainting policies. Attacks usually trigger writes to various log files in the `/var/log` directory and to `/dev/null`. The NoI, NoIA and NoIAN policies generate false dependencies from tainted processes to other processes via these files. If these files are white-listed, the dependency caused by reading these files is ignored, which reduces false positives.

## 4.2 Intra-object Dependencies

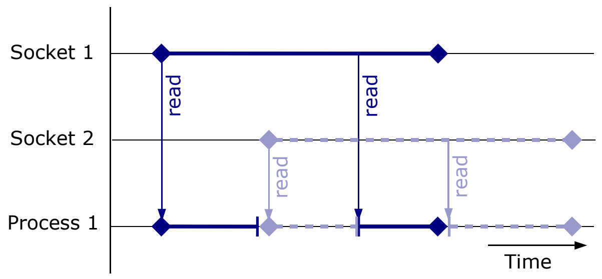
Until now, we have only discussed inter-object dependencies that cause information flow across objects. However, we also consider information flow over time within an object and call it an intra-object dependency. The dependency model described above assumes that intra-object dependencies exist for the lifetime of each object. With this conservative approach, once an object is tainted, it remains tainted forever. For example, a process remains tainted until it dies. This approach avoids missing dependent tainted operations but, unfortunately, it can be too coarse-grained, especially when objects exist for long periods. For example, a server process often performs activities that are logically distinct for each connection. Tainting based on this long running process will generate false dependencies between these unrelated activities simply because the same process issues them.

To reduce these false dependencies, we introduce the notion of intervals to intra-object dependencies so that each interval is considered independent of the other intervals. When an object is tainted, it remains tainted until the end of its current dependency interval, and we name this time period the tainted interval. Below, we describe how we define the dependency intervals for each type of object.

**Processes** For processes, we consider two cases: 1) *server* processes that communicate using a socket connection that is initiated by a remote source, and 2) all other processes. In the latter case, we define only one dependency interval, which is simply the lifetime of the process. In the former case, a server process can be the initiating point for external attacks. Such a process or its parent is typically a demultiplexing point for large numbers of unrelated activities. Hence, creating different intervals for the unrelated activities can significantly avoid false dependency sharing.

We argue that successful socket read operations can distinguish intervals for a server process. A read from a different remote socket indicates that the process switches "context" to work for another unrelated activity. Such a read terminates the current interval and starts another interval. This interval method can be used for various common server models such as separate processes started by Inetd, worker processes that handle different multiple connections and event-driven servers that multiplex the activities of different connections.

Figure 4.1 shows inter-object and intra-object dependencies. Similar to the notation used in Magpie [4], the start and end of an intra-object dependency interval is shown with a ⧫ symbol. However, unlike Magpie that aims to create sets of related objects and uses undirected dependencies, the dependencies shown here are directed. The figure shows two different intervals for Process 1 based on whether the process is serving Socket 1 or Socket 2. Each of these intervals is multiplexed over time as shown by the | symbol.



Objects are shown on the left side of the figure. Inter-object dependencies are shown as vertical arrows while intra-object dependencies are shown as horizontal lines. The start and end of an intra-object dependency interval is shown with a ♦ symbol. Dependency intervals can be multiplexed over time as shown by the | symbol for Process 1. For this process, reading from different sockets starts the different intervals.

Figure 4.1: Inter-object and intra-object dependencies

**File name** With the conservative approach, once a name in a directory is tainted, it remains tainted forever. For example, say a tainted name is removed. If a legitimate process creates the same name in the same directory again, this name still remains tainted. With dependency intervals, a file name interval starts when a new file name is created and it ends when the file name is removed.

**Files content** For file content, the dependency interval starts when a new file is created and it ends when the file is removed. In addition, the interval ends and another interval starts if the data of an object is completely overwritten. For example, the complete truncation of a file starts a new interval since the truncation stops any file content related dependency.

**File attribute** For file attribute, the dependency interval starts when a new file is created and it ends when the file is removed.

**Sockets** For sockets, the dependency interval is simply the lifetime of the socket. Because the states at the remote end of the socket are not available, we can only conservatively define one dependency interval for a socket.

# Chapter 5

# Implementation

The implementation of the analyzer is divided into four components: the preprocessor, the dependency rules, the trace phase and the propagation phase. The preprocessor extracts operations from the Forensix system-call data. The trace phase finds the attack sources and the propagation phase handles the tainting process according to dependency rules. Figure 5.1 shows an overview of the implementation.

## 5.1 Preprocessor

The semantics of system calls can be complex. For example, many system calls take a pathname as an input argument. The kernel internally translates the input pathname to a kernel file object. During the translation, the kernel reads the name, permission and ownership of each of the components of the path to locate the kernel file object as well as to determine access privileges. Some system calls allow an input flag to control their behaviour. For example, the `O_CREAT` flag controls whether an `open` system call will create a new file if the file does not exist.

The preprocessor interprets the complexity of system calls and maps them to one or more simple operations defined in Table 3.1. It mimics some of the kernel's behaviour in terms of handling system calls. The detailed mapping is shown in Appendix A. While some system

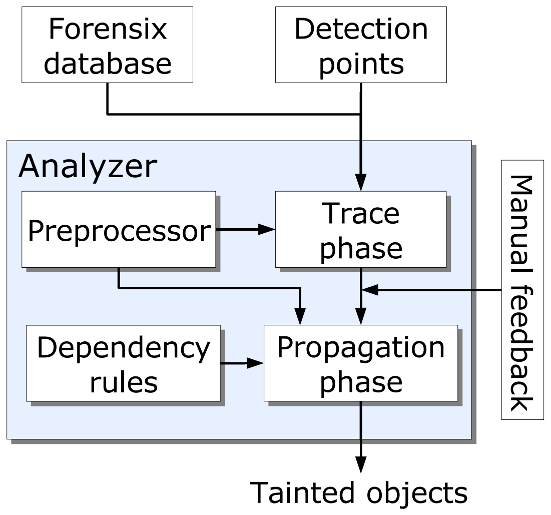


Figure 5.1: Overview of the Taser analyzer implementation

calls, like `fork`, have simple one-to-one mapping, some system calls have complex mappings. For example, the `rename` system call takes an old path and a new path as arguments. If the new path exists, it will be atomically replaced. This system call translates into the following operations: 1) read file name and attribute for all the components in the old path (to access the old pathname), 2) read file name and attribute for all the components in the existing new path (to access the new pathname), 3) write file name of the existing new path (to remove the name), 4) write file content and attribute of the existing new path if it is the last link (to remove the kernel file object), 5) write file name of the existing old path (to remove the name), 6) write file name of the new path (to create the new name).

## 5.2 Dependency Rules

The dependency rules consist of the analyzer enhancements: tainting policies, white list, and dependency intervals. Based on these enhancements, this component makes two decisions: 1) whether to start a tainted interval for a sink object, or 2) whether to end the current tainted interval for a sink object. Without these enhancements, the first decision is always true and the second decision is always false.

A tainted interval is started when a dependency caused by an operation is considered by the tainting policy and when the source object is not in the white list. A tainted interval ends when dependency intervals are enabled and the conditions described in Section 4.2 are met.

The implementation of dependency rules is straightforward except for the complications that arise in identifying server processes. Previously, we defined a server process as one that communicates on a socket connection that is initiated by a remote host. However, this process may fork child processes to run utility programs that also use the same socket connection. For instance, an Apache process forks a child to execute a common gateway interface (CGI) program. These utility processes should not be treated as server processes. Fortunately, the parent process of a real server process does not communicate with the remote host on the connection. The parent simply accepts the connection and lets the server process service the request. As a result, an additional constraint, that the parent process should not read from or write to the same connection as a server process, is needed to identify server processes.

Of the policies listed in Table 4.1, the snapshot policy and the conservative policy are not implemented as part of dependency rules at the moment. The snapshot policy finds the file-system objects that would be reverted by restoring a snapshot. We implement this policy in an ad hoc manner by tainting all process objects from the time of the snapshot and then running the analyzer with the NoI policy. The conservative policy cannot be reliably implemented because the Forensix auditor does not currently capture all IPC-related information.

## 5.3 Trace and Propagation Phases

The implementation of the trace and the propagation phases is similar. The trace phase consists of three steps. It first queries the auditor to get the system-call data, and then in the second step, it queries the preprocessor to obtain the operations for each system call. Finally, the trace phase considers each operation in reverse time order to determine the tainted source objects. When the sink object of an operation is in a tainted interval, this step taints the source object by creating a tainted interval that starts from the beginning of time ($-\infty$) until the time of the operation. This tainted interval is used if this source object becomes a sink object in another operation. To avoid missing attack sources, this phase is conservative and ignores the dependency rules.

The propagation phase also consists of three steps. The first two steps of the trace and the propagation phases are exactly the same. In the last step, the propagation phase considers each operation in forward time order to determine the tainted sink objects. When the source object of an operation is in a tainted interval, this step queries the dependency rules to determine whether to create a tainted interval for the sink object. The tainted interval is created from the time of the operation to the end of time ($\infty$). In addition, if the dependency rules component decides to end the tainted interval at an operation, the propagation phase limits the end time of the tainted interval to the time of the operation.

# Chapter 6

# Evaluation

We evaluate the analyzer using two criteria: 1) the accuracy with which it correctly identifies attack-related objects and 2) the performance of the analyzer. The analyzer can only be used in conjunction with the Taser auditor. As a result, we also evaluate the performance and storage cost of using Forensix as the auditor.

The experimental setup consists of a target and a backend machine. The target machine is an AMD Athlon 2600+ machine with 512 MB memory. It runs stock Redhat 7.2 together with the Forensix logger. It contains four vulnerable services or executables: the samba and the wu-ftpd daemon that allow remote root exploits, and the sendmail and the pwck-setuid programs that allow local root escalation exploits. The backend machine is an Intel Pentium 4 2.4 GHz CPU with Hyper-Threading and 512 MB memory. It runs Redhat Fedora Core 3 and uses the MySQL version 4.1.10 as the Forensix database. The target system is a server that provides an online album service using a web application called Gallery.

The experiment is run for approximately a week. During this time, a third machine runs a program called Galhogger to simulate user interactions against the Gallery service continuously. Galhogger simulates an anonymous user that browses the albums every 2 seconds on average, and five registered users that each modifies the albums every 3000 seconds on average to generate concurrent activities and to load the target system.

## 6.1 Accuracy

We use the entire Taser recovery system to evaluate the accuracy of the analyzer. The output of the analyzer is used by Taser to generate a list of actions that revert the tainted operations performed on tainted objects. Since the analyzer is the only component that decides whether objects are tainted, the Taser output reflects the decisions made by the analyzer and therefore its accuracy. This list contains the minimal number of recovery actions and is more compact than the analyzer output.

This evaluation consists of six scenarios. Each scenario represents a different flavour of situation in real life that requires file-system recovery. Four of the scenarios are intrusion-related and two reflect system mis-configuration cases.

As described in Section 3.2, the analyzer first runs the trace phase to obtain the attack source from detection points and then the propagation phase. The trace phase is run without ignoring any dependency so that it finds all potential attack sources. The propagation phase is run with five different policies: Snapshot, NoI, NoIAN and NoIANC. The NoIA policy has results similar to the NoI policy and is not shown in this evaluation. The output of the propagation phase is then fed to the recovery system to generate the set of recovery actions.

Two metrics are used to evaluate the accuracy: 1) the number of false positives, which are recovery actions generated to undo legitimate operations, and 2) the number of false negatives, which are the attacker's operations that are not subject to recovery. Recovery operations for each file content object are aggregated as a single recovery action, whereas for name and attribute objects, each recovery operation is counted as a recovery action. To calculate the number of false positives and negatives, the correct number of recovery actions is determined manually. To evaluate the accuracy over time, Taser is run one day and one week after the incidents in the scenarios.

The Snapshot policy is presented as a base case for comparison. This policy can only have false positives, as all operations happen after the snapshot time are reverted. Note that illegitimate actions of all scenarios are performed prior to the Taser's runs. As a result, the false positive numbers for the snapshot policy are similar across scenarios.

The NoI and the NoIAN policies create dependencies due to reading file content. In general, they can generate large numbers of false positives due to false dependencies caused by reading character device files and log files. A small set of files are white-listed to minimize the number of false positives. Unless explicitly mentioned, the default white list for these policies consisted of the following: 1) all character device files such as `/dev/null` and `/dev/pts/0`; 2) the log files `/var/log/wtmp`, `/var/log/wtmp.*`, `/var/run/utmp`, `/var/log/lastlog`; and 3) shell history files. Note that white-listed files are irrelevant to the Snapshot and the NoIANC policies because the former already has all process objects tainted and the later does not concern about content to process dependencies.

By default, dependency intervals defined in Section 4.2 are enabled for all the policies. One of the scenarios used a server process, and in this scenario, results that have dependency intervals enabled and disabled are compared to show that disabling dependency intervals for tainted worker processes can cause a large number of false positives.

The following subsections describe of the scenarios and their results.

| | | Analysis done one day after attack | | | | Analysis done one week after attack | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Scenario** | **Recovery Actions** | **Snapshot** | **NoI** | **NoIAN** | **NoIANC** | **Snapshot** | **NoI** | **NoIAN** | **NoIANC** |
| Illegal storage | 507 | 633, 0 | 2, 0 | 0, 0 | 0, 0 | 4154, 0 | 7, 0 | 0, 2 | 0, 2 |
| Content destruction | 739 | 1877, 0 | 0, 0 | 0, 0 | 0, 0 | 5338, 0 | 0, 0 | 0, 0 | 0, 1 |
| Unhappy student | 167 | 1106, 0 | 2, 0 | 0, 0 | 0, 1 | 4617, 0 | 4, 0 | 0, 0 | 0, 1 |
| Compromised database | 3 | 814, 0 | 0, 0 | 0, 0 | 0, 2 | 2557, 0 | 0, 0 | 0, 0 | 0, 2 |
| Software installation | 350 | 1542, 0 | 1, 0 | 0, 0 | 0, 0 | 5006, 0 | 1, 0 | 0, 0 | 0, 0 |
| Inexperienced admin | 39 | 1366, 0 | 11, 0 | 11, 0 | 0, 0 | 4982, 0 | 11, 0 | 11, 0 | 0, 0 |
| Inexperienced admin (dependency intervals disabled) | 39 | 1366, 0 | 415, 0 | 415, 0 | 125, 0 | 4982, 0 | 701, 0 | 701, 0 | 126, 0 |

For each scenario, the second column shows the correct number of recovery actions as determined manually. The rest of the columns show the accuracy of four dependency policies in terms of false positives and false negatives, separated by commas. The accuracy of the policies are shown one day and one week after the attack.

Table 6.1: Analyzer accuracy using various dependency policies

### 6.1.1 Scenarios

**Illegal storage:** A user logs into the system and launches the pwck local escalation exploit to get the root shell and creates a new root account root100 by directly writing to the `/etc/passwd` and the `/etc/shadow` files. This attacker creates a directory under another user's directory (as root) and downloads 500 illegal pictures into this directory. Finally, he downloads a trojaned binary `ls` program in the user's bin directory to hide the existence of the illegal directory. Later, the victim user logs in, uses the trojaned `ls` program and creates two files in his home directory. The attacker logs back in as root100 after two days and downloads two more pictures into the hidden directory.

**Correct recovery actions:** Remove all the illegal pictures and the hidden directory, the trojaned `ls` binary, and the home directory of the attacker's root100 account. In addition, the legitimate versions of the `/etc/passwd` and `/etc/shadow` files need to be recovered.

**Detection points:** The trojaned `ls` program is detected by the victim and given to the trace phase.

**Results:** The trace phase detects the remote connection of the attacker and propagation is started using the remote host address. The NoI and the NoIAN policies with the default white list generated several false positives because `/etc/passwd` and `/etc/shadow` are written by the attacker and their content is read by all the following ssh login processes which also get tainted. As a result, all activities in the these login sessions are to be reverted by the recovery system. The content objects for `/etc/passwd` and `/etc/shadow` are then added to the white list and the analysis is run again. Table 6.1 shows that two false positives for the NoI policy when Taser is run after a day. These errors are caused by the victim's shell process that gets tainted as it accesses the name of the trojaned `ls` program. Therefore, the two new files created by him are tainted and will be removed during recovery. The other two policies have no errors when recovery is performed after one day.

When Taser is run one week after the attack, the NoIAN and the NoIANC policies have two false negatives because the attacker's second login is not caught tainted as the password files are put into the white list. The two new picture files created in this session are missed by these policies. The NoI policy has no false negatives because the second login by the attacker accesses the tainted root100 directory that taints this session. Table 6.1 shows that the recovery results after a week are similar to the recovery results after a day, and therefore, for the rest of the scenarios, the discussion will be based on the recovery results after a day.

**Content destruction:** A software developer has been working on the files `src/project.c`, `hfiles/p1.h` and `hfiles/p2.h`. He has also saved a backup of the project.c file in `backup/project.c.bak`. Another developer on the system launches the sendmail local escalation exploit to get the root shell. This attacker deletes the `project.c` and `p2.h` files. The victim notices that the `project.c` file is missing. He copies the backup file to the `src` directory and also moves the `p1.h` file to the `src` directory. Then, he deletes the `hfiles` directory and notifies the administrator.

**Correct recovery actions:** Remove numerous files generated by the sendmail attack, restore the deleted `p2.h` file in the `hfiles` directory, recover the original `project.c` file and deal with different versions of this file.

**Detection points:** The missing `p2.h` and `project.c` files.

**Results:** The trace phase detects the attacker's login process, which is used for propagation. The results show that none of the policies, except the snapshot policy, have any errors. The `hfiles` directory is created along with the recreation of the `p2.h` file. The original `project.c` file is recovered with an extra file name extension to differentiate from the existing version.

**Unhappy student:** An attacker launches a remote attack on the wu-ftpd daemon running on the system and modifies the permissions of a grades file in a professor's home directory to be globally writable. Later, student A (an accomplice) with a regular account modifies the grades file in the professor's directory and also copies the professor's whole home directory in his own directory. Then, student B (another accomplice) logs in and copies the modified grades file to his home directory and creates two other files.

**Correct recovery actions:** Recover the original grades file in the professor's directory, restore the attribute of this file, and remove all copied files in both student A's and student B's home directories.

**Detection points:** The grades file that the professor finds is writable by others.

**Results:** The trace phase detects the remote attacker's root shell as well as student A's login session but not student B's login because B did not modify the grades file. The NoI propagation policy detects and taints student B's shell process and the two files created by him which are false positives. The NoIANC policy on the other hand did not taint student B's operations, which leads to a false negative because the illegal copy of the grades file in student B's home directory is not removed. The NoIAN policy had no errors because this policy tainted B's copy operation but not B's entire shell process.

**Compromised database:** Authenticated MySQL clients update a MySQL database running on a remote server. An attacker launches a remote attack on the Samba daemon running on the target system, gets a root shell and creates an SSH backdoor by writing his public key to root's `authorized_keys2` file. Later, other remote legitimate clients insert transactions into the database. After six hours, the attacker uses the ssh backdoor to log back into the machine. He issues a local MySQL query to remove some tuples from the database. After that, more legitimate clients update the database.

**Correct recovery actions:** Remove the attacker's ssh backdoor by removing his public key from the `authorized_keys2` file. In addition, recover two files associated with a MySQL table in the compromised database.

**Detection points:** Snort [21] detects the Samba attack but is unable to give further information regarding the damages done at the target machine. So the Forensix tools [11] are used to determine the attacker's root shell session as the detection point.

**Results:** The trace phase starts from the root shell and detects the attacker's first connection, which is used for propagation. The results show that none of the policies have any false positives. All of them recover the `authorized_keys2` file. Using the NoI and the NoIAN policies, the analyzer detects the attacker's second login via the dependency caused by the tainted `authorized_keys2` file. Taser then restores the modified database files to the state right before the attacker's second login session in which he modifies the database. The rest of the legitimate database updates are not recovered as the database's file content objects are tainted since the attack. The Snapshot policy would revert the database to a state before the attacker's first login and miss *all* database writes since then. The file-based recovery approach works in this scenario because MySQL is configured not to use transactions. For transactional databases, the database recovery logs would need to be incorporated in the recovery process [17, 19]. The NoIANC policy has two false negatives because it misses the attacker's second login as well as the delete query.

**Software installation:** Unlike the previous scenarios, the next two scenarios present and analyze system administration errors. Using a root account, an administrator installs RealPlayer 8 in the wrong directory which causes it to create many files and directories in this directory. In addition, it creates or updates various configuration files of Netscape, KDE and Gnome, and creates directories in `/root` such as `.netscape/plugins`. Later, the root user browses the web with the netscape browser and downloads and saves a PDF reader plugin for Netscape in this directory.

**Correct recovery actions:** All the RealPlayer files and directories should be removed and the configuration files should be restored.

**Detection points:** One of the RealPlayer files.

**Results:** The trace phase detects the process of the RealPlayer installer program, which is used for propagation. For this scenario, none of the policies generated any false negatives. However, the NoI policy has one false positive because the Netscape plugins directory is tainted and this policy taints the PDF reader plugin that the recovery later removes. The NoIAN and the

NoIANC policies do not taint the PDF reader plugin. Although the plugins directory is tainted, it remains in the system because it contains the legitimate plugin file.

**Inexperienced administrator:** The administrator uses the photo Gallery software (which is also used as background load in the experiments) to store his digital pictures and also creates an account for a guest user. The new account is set up with a weak password because the administrator expects the guest to change the password soon. Then, the administrator adds new albums and pictures under his account and logs off. Before the guest can change his password, an attacker at a remote site logs into the guest's account by using a dictionary attack. The attacker creates two new albums and uploads 14 pictures to the target machine and then views the administrator's albums. Later, the administrator views the albums and discovers inappropriate images in the two new albums. He contacts the guest user and finds out that the guest user is not responsible for these albums.

**Correct recovery actions:** Remove the attacker's albums and all related data (such as thumbnails) generated by Gallery.

**Detection points:** A directory that contains one of the attacker's albums.

**Results:** The trace phase detects the attacker's remote connection and propagation is started using the remote host address. Gallery maintains album, image, thumbnail and photo visit counters in a file hierarchy. Table 6.1 shows that there are 39 necessary recovery actions: 2 of which are to remove the attacker's albums, 14 are to remove the images, 16 are to remove the automatically generated thumbnails, 6 are to remove the album data files generated by Gallery (such as `album.dat`), and the remaining one is to recover a common data file called `albumdb.dat` that contains some global Gallery information. The NoI and the NoIAN policies generated many false positives because these policies create content-related dependencies and Gallery always reads the `albumdb.dat` file which taints all connections and their subsequent operations. After the `albumdb.dat` file and the per-directory `album.dat` files are added to the white list, running Taser with these policies generated 11 false positives. In contrast, the NoIANC policy generated no false positives because the tainted status does not propagate via the `albumdb.dat` file.

Gallery runs on the Apache server that uses a worker model for servicing requests. The experiment above is performed using multiple tainted intervals, one per connection, for the Apache worker processes. The same analysis is run again but allowing only a single tainted interval for the worker processes for this run. The last row of Table 6.1 shows that the single interval setting results in false positives even with the NoIANC policy, the most optimistic policy, because once a worker process is tainted by the malicious connection then it remains tainted even if it later services a legitimate connection. If it is not that case that Apache kills and re-spawns a new worker thread regularly, the single interval setting will cause all Gallery actions after the attack to be reverted.

### 6.1.2 Discussion

The previous subsection has evaluated the accuracy of the analyzer in terms of correctly recovering legitimate data under various scenarios. There are several key results from Table 6.1 that are worth highlighting. First, the table shows that the analyzer typically achieve high accuracy. The number of false positives or negatives remains small comparing to the number of recovery actions required.

Second, the accuracy results do not vary significantly when Taser is run one day or one week after the attack. Because the usefulness of Taser does not degrade over time, the timeliness requirement for intrusion detection systems can be relaxed.

Finally, the most significant result is that no tainting policy performs ideally under all circumstances, a result that may seem undesirable. However, note that the optimistic policies have relatively few false positives while the conservative policies have few false negatives. For example, the NoIANC policy had no false positives in all scenarios while the NoI policy had no false negatives. By comparing the output of the different policies, one can quickly determine the correct set of recovery actions. Recovery actions that occur in all policies are most likely to be correct actions. The difference in the outputs of these policies are the recovery actions that are ambiguous, that is, whether these actions are legitimate or not. This difference, in terms of the sum of false positives and false negatives of the scenarios, is indeed a small number. Hence, it should be easy to classify these actions manually.

## 6.2 Performance Measurements

Taser utilizes Forensix to provide an audit log of all system-call data relevant to file-system recovery. This section shows the performance of the analyzer and the cost of using Forensix as the auditor. Note that the target and the backend performance are mostly decoupled. For example, the analyzer that is run at the backend machine has relatively little impact on the performance of the target machine.

### 6.2.1 Analyzer

Table 6.2 shows the time required to run the analyzer when it is started one day and one week after the attack. For each of the four dependency policies, the average analyzer runtime and the 95% confidence intervals in minutes are shown. The average is taken across all the scenarios that are evaluated in Section 6.1.

The trace phase was run for each scenario once and the attack time interval is set to one hour. It takes less than a minute for every scenario and is not shown in the table. The propagation phase takes below 7 minutes when the analyzer is started one day after the attack. However, the propagation time can be as much as one hour when the analyzer is run after a week. Note that the propagation time grows with the time to analysis because propagation visits every operation done by the target system to determine dependencies.

The performance of the propagation phase is indeed held back by the performance of the Forensix database. Approximately 80% of the propagation time is spent in the MySQL database retrieving system-call data sorted in time order. While using a database to store the

| Policy | One day | One week |
|---|---|---|
| Snapshot | 5.5±1.1 | 102.9±7.4 |
| NoI | 4.5±0.9 | 58.3±3.3 |
| NoIAN | 4.4±1.0 | 56.2±3.1 |
| NoIANC | 4.0±0.9 | 63.6±12.6 |

The performance numbers of the analyzer run one day and one week after the attack. Each number is shown in minutes and averaged across the different scenarios.

Table 6.2: The analyzer performance

Forensix audit log is useful for general intrusion analysis, we postulate that the analyzer can be greatly optimized by using a customized implementation for data storage and retrieval.

While the performance results above show that the analyzer can be run relatively quickly, note that the administrator must still spend time choosing the appropriate attack source objects between the trace and the propagation phases of the analysis. Interactive and graphical analysis tools that can display the tainted source objects together with their attribute, such as the number of dependent objects, can ease this process.

### 6.2.2 Auditor

Auditing system calls imposes overhead on the target system while loading this data into a database at the backend machine imposes overhead on the backend. With the load imposed by Galhogger on the Gallery photo album application, the logging overhead at the target is insignificant. The cost imposed at the backend, averaged per day, is shown in Table 6.3. The system load and hence the daily numbers do not vary much across different days. The table shows the number of system calls generated on the target machine. The most common system calls consist of `read` (5.8 M), `open` (3.0 M), `close` (2.0 M), `mmap` (1.7M), `write` (371 K), `dup` (155 K), signal (106 K), `connection` (31 K), `unlink` (16 K), `exec` (9 K) and `fork`

| | |
|---|---:|
| Number of operations | 13.3 Million |
| Size of flat files | 1.9 GB |
| Size of database | 2.3 GB |
| Database loading time | 36.3 min |

Table 6.3: Average daily backend statistics

(8 K). These constitute 99% of all operations.

The total amount of uncompressed file data generated is 1.9 GB per day. When this data is loaded into a database, the database size grows by 2.3 GB per day. The loading time is 36.3 minutes per day. Another way to interpret this result is that the backend system can sustain loads that are approximately 40 $(24 \times 60 \div 36.3)$ times larger than the load imposed by Galhogger or one backend system can audit 40 target machines with the same load. The database loading time is the main bottleneck in Forensix. A Forensix audit log implementation optimized for the analysis could potentially avoid the loading times.

**Target Performance Under Heavy Load**

To measure the overhead of auditing at the target system under heavy loads, two benchmarks are used: Webstone and Linux kernel build. The Webstone benchmark stresses a standard Apache web server running on the target system by issuing back-to-back client requests and is a representative of a loaded server environment. A third machine, which has the same configuration as the target machine, runs as the Webstone client and it is connected to the target machine with a Gigabit network. The kernel build benchmark determines the overhead imposed for CPU-bounded applications in a regular desktop environment.

The performance overhead of auditing on the target machine is measured by logging the system calls on the target machine and streaming these data to the backend where they are stored in append-only files. Auditing has an insignificant effect on kernel compilation (0.6%) while Webstone throughput decreases from 258.3 Mbs to 239.2 Mbs or about 7.4%. These results are encouraging because they show that even under heavy load, the Forensix logging mechanism on the target has low overhead.

The amount of compressed data collected in these experiments (extrapolated per day) is 8 GB per day for kernel compilation and 11 GB per day for Webstone. Even though the amount of logged data is roughly the same, the Webstone throughput suffers much more than kernel compilation because Webstone generated approximately 11 times the number of system calls compared to kernel compilation. While the storage requirements of Forensix can be large under heavy loads, the large amount of network capacity and massive and inexpensive storage space available in local networks today (for example, a terabyte costs between $500 and $1000) render this approach feasible for reliable intrusion analysis.

# Chapter 7

# Related Work

The goal of this thesis is to perform attack analysis to enable recovery after an intrusion. In this chapter, we first describe related work in the areas of intrusion analysis and causality-based analysis. Then, we describe related research in recovery, in particular, database recovery. Finally, we discuss complementary computer security research related to intrusion detection and sandboxing.

## 7.1 Intrusion Analysis

The analyzer is directly motivated by the work on backtracking intrusions [16]. Their BackTracker system uses a time-based approach to generate dependencies between processes, files and sockets and uses a dependency graph to view intrusions. The primary difference between the two systems stems from the difference in their goals. While BackTracker is focused on tracking the sources of an intrusion, the analyzer generates a set of tainted files that need to be recovered. As a result, the BackTracker's taint analysis policies are conservative or else it would miss the intrusion, while the analyzer provides optimistic policies so that legitimate data can be preserved as much as possible during recovery. BackTracker considers each file as a single object, while the analyzer separates the content, name, and attribute of a file into different objects. This difference in granularity allows the analyzer to provide optimistic policies. Furthermore, unlike BackTracker, the analyzer uses interval-based analysis to improve its accuracy.

Sebek [15] provides a honeypot[1] environment to capture activities of attackers and stores this data in a separate machine for administrators to review the intrusion session. Sebek has a similar architecture as the Forensix auditor but it only audits `write` system calls. This is sufficient for Sebek because it attempts to reveal data such as keystrokes, uploaded files and passwords, whereas Forensix provides a framework for general intrusion analysis. Another difference between Sebek and Taser is that Sebek assumes a honeypot environment where all activities on the target machine are assumed to be related to an intrusion. However, Taser needs the analyzer to determine whether an activity is intrusion-related for a general system.

## 7.2 Causality Analysis

Magpie [4] extracts the control flow and the resource requirements of requests in a clustered server environment by monitoring kernel and application-level events. Then it correlates these events using an application-specific event schema. Magpie uses interval-based correlation similar to the analyzer's dependency intervals. However, while Magpie uses undirected dependencies to cluster events, the analyzer uses directed dependencies to derive data flow.

Project5 [2] exposes performance problems in a distributed, multi-layer system by means of a graph of communicating nodes. The nodes and edges in the graph are assigned latencies, and performance bottlenecks are determined by calculating the total latency of critical paths. Because Project5 does not know the semantics of the distributed application, it uses statistical techniques to infer dependencies. The Taser analyzer knows the semantics of the system operations and therefore it can precisely determine the dependencies.

Data lifetime analysis using system-level simulation [6] or hardware-based information flow [30] allows detecting or protecting programs against malicious attacks by identifying spu-

[1]Honeypot is a system that masquerade as abusable resources to attract intruders.

rious information flows from untrusted I/O sources. Both can provide more accurate taint analysis than the analyzer's approach but either run orders of magnitude times slower or require special architectural support.

## 7.3 Database Intrusion Recovery

Fastrek [19] recovers databases by attributing modifications to malicious activities and then rolling back changes selectively. A potential issue with this approach is cascading aborts where a legitimate operation is rolled back if it may have depended on the data produced by a tainted operation. The conservative policies used by the analyzer exhibit similar problems. However, it is not clear whether optimistic policies are possible with transactional databases.

## 7.4 Intrusion Detection

System-call traces have been used in the past to identify normal system behavior and then to automatically detect suspicious behavior or intrusions [14, 26, 27, 8]. These approaches examine system-call patterns over a short window of 5-100 calls. Our tainting analysis approach correlates both system-call operations and their arguments, which could be used to implement more accurate intrusion detection algorithms.

RIPPER [1] correlates application- and system-level logs on the target system to detect intrusions. We argue that these logs can be destroyed by the attacker, and hence they are not suitable for analysis. However, since Taser is capable of recovering files after an intrusion, it should be possible to use RIPPER to analyze attacks using log files that are recovered by Taser.

## 7.5 Sandboxing

Sun [31] provides a safe execution environment (SEE) that enables users to try out new software (or configuration changes to existing software) without fear of damaging the system in any way. This is accomplished via a novel one-way isolation mechanism where processes running within the SEE are given read-access to the environment provided by the host OS, but their write operations do not affect the host until a commit point. The commit is performed if a consistency criteria is met or else the SEE is rolled back. This approach allows recovery only until the commit point. Furthermore, rollback caused by violating the consistency criteria can become more likely for long running SEEs.

Systrace [20] notifies the user about system calls executed by an application. Then it generates a sandboxing policy based on user response. Systrace relies on pathname to identify files. This makes the policies prone to race attack because the same name may point to different files at different times. Sandboxing raises the issue of policy selection, i.e, determining what actions are permissible for a given piece of software.

# Chapter 8

# Conclusions

Today, snapshot-based file-systems are typically used to recover from intrusions or human errors. This approach is well understood but it works well only when intrusions or errors can be immediately detected. Otherwise, a snapshot before an attack loses legitimate user modifications that occurs after the attack. The Taser intrusion recovery system addresses this shortcoming of snapshots. The key problem that this thesis solves is to accurately determine the set of tainted file-system operations so that they can be reverted.

We evaluated the accuracy of Taser in dealing with a wide range of intrusions as well as erroneous user activity scenarios. Our evaluation shows that our most optimistic analysis policy does not taint legitimate data but can miss intrusion activity, while the more conservative policies avoid missing any intrusion activity but require some hand tuning by white-listing files. Our experience with Taser shows that an appropriate set of recovery actions can be determined quickly when the results of the different policies are compared. We believe that Taser provides the basis for developing automated intrusion recovery solutions.

## 8.1 Future Directions

Our tainting analysis approach can be applied to improve anomaly detection tools. Current anomaly detection tools create a profile of normal system use by limiting themselves to monitoring activities per process. The tainting approach can be used to build profiles based on monitoring activities across processes as well as file-system and network activities.

We also wish to explore if the tainting analysis approach can be used for intrusion recovery of transactional databases. Transactional databases have four features, atomicity, consistency, isolation, and durability, to protect the integrity of data. However, these features are not guaranteed at the system-call level at which Forensix audits the system. Considering data from both the transaction log of the database and the audit log of Forensix may allow the analyzer to determine whether there are any transactions performed by an attacker and possibly revert these tainted transactions.

# Bibliography

[1] Cristina Abad, Jed Taylor, Cigdem Sengul, Yuanyuan Zhou, William Yurcik, and Ken Rowe. Log correlation for intrusion detection: A proof of concept. In *Proceedings of the Annual Computer Security Applications Conference*, pages 255–265, December 2003.

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 74–89, October 2003.

[3] Edward C. Bailey. *Maximum RPM*. Sams, August 1997.

[4] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 259–272, 2004.

[5] CERT Coordination Center. Cert/cc statistics 1988-2004. `http://www.cert.org/stats/cert_stats.html`.

[6] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, pages 321–336, August 2004.

[7] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, December 2002.

[8] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[9] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[10] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[11] Ashvin Goel, Wu-chang Feng, David Maier, Wu-chi Feng, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. In *Proceedings of the International Workshop on Security in Distributed Computing Systems (SDCS)*, June 2005. In conjunction with the International Conference on Distributed Computing Systems (ICDCS).

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

[13] Bobbie Harder. Microsoft windows system restore. `http://msdn.microsoft.com/library/en-us/dnwxp/html/windowsxpsystemrestore.asp`, April 2001.

[14] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[15] The Honeynet Project. Know your enemy: Sebek. `http://www.honeynet.org/papers/sebek.pdf`.

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

[17] Peng Liu, Paul Ammann, and Sushil Jajodia. Rewriting histories: Recovering from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[18] Nicholas Petreley. Security report: Windows vs Linux. The Register, October 2004. `http://www.theregister.co.uk/security/security_report_windows_vs_linux`.

[19] Dhruv Pilania and Tzi cker Chiueh. Design, implementation, and evaluation of an intrusion resilient database system. Technical Report TR-124, SUNY, Stony Brook, April 2005.

[20] N. Provos. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium*, pages 257–272, August 2003.

[21] Martin Roesch. Snort - Lightweight intrusion detection for networks. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, pages 229–238, November 1999.

[22] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[23] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 110–123, December 1999.

[24] sd and devik. Linux on-the-fly kernel patching without LKM. Phrack issue 58, December 2001.

[25] Secunia. Secunia vulnerability report. `http://www.secunia.com`.

[26] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the USENIX Security Symposium*, pages 63–78, August 1999.

[27] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the USENIX Security Symposium*, pages 185–198, August 2000.

[28] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 43–58, 2003.

[29] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 165–180, 2000.

[30] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGARCH Computer Architecture News*, 32(5):85–96, 2004.

[31] Weiqing Sun, Zhenkai Liang, R. Sekar, and V.N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.

[32] Andy Watson and Paul Benn. Multiprotocol Data Access: NFS, CIFS, and HTTP. Technical Report TR3014, Network Appliance, Inc., 1999. `http://www.netapp.com/tech_library/3014.html`.

[33] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the USENIX Security Symposium*, pages 17–31, 2002.

[34] Huagang Xie and et. al. Linux intrusion detection system (LIDS) project. `http://www.lids.org/`.

# Appendix A

# Mapping for System Calls to Operations

| System call | Operations |
|---|---|
| open, creat | 1) read file name and attribute of all path components<br>2) write file name and attribute of the file if a file is created |
| execve | 1) read file name and attribute of all path components<br>2) executes the file |
| mkdir | 1) read file name and attribute of all path components<br>2) write file content, name and attribute of the directory |
| unlink | 1) read file name and attribute of all path components<br>2) remove file name of the last path component<br>3) remove file content and attribute file if it is the last link |
| mknod | 1) read file name and attribute of all path components<br>2) write file name of the last path component<br>3) write file content and attribute of the file |
| rmdir | 1) read file name and attribute of all path components<br>2) remove file content, name and attribute of the directory |

| | |
|---|---|
| chown,<br>chown32 | 1) read file name and attribute of all path components<br>2) write file attribute of the file |
| lchown,<br>lchown32 | 1) read file name and attribute of all path components<br>2) write file attribute of the file |
| fchown,<br>fchown32 | 1) write file attribute of the file |
| chmod | 1) read file name and attribute of all path components<br>2) write file attribute of the file |
| fchmod | 1) write file attribute of the file |
| symlink | 1) read file name and attribute of all path components of the old and new path<br>2) write file name of the last path component of the new path<br>3) write file content and attribute of the new file |
| link | 1) read file name and attribute of all path components of the old and new path<br>2) write file name of the last path component of the new path |
| rename | 1) read file name and attribute of all path components of the old and new path<br>2) remove file name of the last path component of the existing new path<br>3) remove file content and attribute of the existing new file if it is the last link<br>4) write file name of the last path component of the new path<br>5) remove file name of the last path component of the old path |
| truncate,<br>truncate64 | 1) read file name and attribute of all path components<br>2) write file content of the file |
| ftruncate,<br>ftruncate64 | 1) write file content of the file |
| chdir | 1) read file name and attribute of all path components |
| chroot | 1) read file name and attribute of all path components |

| | |
|---|---|
| socketcall | 1) write file content of the socket for send and sendto calls<br>2) read file content of the socket for recv and recvfrom calls |
| read, pread, readv | 1) read file content of the file |
| write, pwrite, writev | 1) write file content of the file |
| fork, clone, vfork | 1) fork |
| sendfile | 1) read file content of the file<br>2) write file content of the socket |