Principles-Driven Forensic Analysis

Sean Peisert
Dept. of Computer Science and Engineering
San Diego Supercomputer Center
University of California, San Diego
La Jolla, CA 92093-0404
sean@cs.ucsd.edu

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562
bishop@cs.ucdavis.edu

Sidney Karin
Dept. of Computer Science and Engineering
San Diego Supercomputer Center
University of California, San Diego
La Jolla, CA 92093-0404
skarin@cs.ucsd.edu

Keith Marzullo
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404
marzullo@cs.ucsd.edu

ABSTRACT

It is possible to enhance our understanding of what has happened on a computer system by using forensic techniques that do not require prediction of the nature of the attack, the skill of the attacker, or the details of the system resources or objects affected. These techniques address five fundamental principles of computer forensics. These principles include recording data about the entire operating system, particularly user space events and environments, and interpreting events at different layers of abstraction, aided by the context in which they occurred. They also deal with modeling the recorded data as a *multi-resolution*, finite state machine so that results can be established to a high degree of certainty rather than merely inferred.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses), Unauthorized access (e.g., hacking, phreaking)*

General Terms

Security, Management, Design

Keywords

Forensics, forensic analysis, forensic principles, logging, auditing, covert channels, compilers, virtual machine introspection, multi-resolution forensics, abstraction shortcuts, race conditions

1. INTRODUCTION

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

-Sherlock Holmes, *A Scandal in Bohemia*, Sir Arthur Conan Doyle (1891)

Forensic analysis is the process of understanding, re-creating, and analyzing events that have previously occurred. *Logging* is the recording of data that will be useful in the future for understanding past events. *Auditing* involves gathering, examining, and analyzing the logged data to understand the events that occurred during the incident in question [2]. The data gathered may also involve decompiling binaries or recovering other remaining evidence, such as saved memory images.

Successful forensic analysis requires the ability to re-create any event regardless of the intent of the user, the nature of the previous events, and whether the cause of the events was an illegitimate intruder or an authorized insider. The ability to do this has progressed very little since the first research on the subject in 1980 [1]. Examples of events that current tools cannot accurately identify include anything that a program is reading or writing to memory. This covers a huge number of possible events and exploits, such as changes to the user environment, covert channels, buffer overflows, and race conditions.

Current techniques to resolve these problems result in generation of too much information, leading to impractical performance slowdowns and high storage requirements. They also suffer from a disparity between the goals of system designers, administrators, and forensic analysts. Five principles address current failures. We believe that any tool that does not follow all of these principles will fail to record actions or results in enough detail to understand their meaning. Current tools use techniques that address some, but not all, of these principles. Thus, they fail to recognize and report many scenarios, or do so incorrectly.

No widely-used operating system records every event, its cause, and its result. Hence we need other ways to generate this information. Three key bases for computer forensics will help us interpret events correctly:

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

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

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

Principle 2: *Assumptions about expected failures, attacks, and attackers should not control what is logged. Trust no user and trust no policy, as we may not know what we want in advance.*

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

Principle 4: *Context assists in interpreting and understanding the meaning of an event.*

Principle 5: *Every action and every result must be processed and presented in a way that can be analyzed and understood by a human forensic analyst.*

Several of these principles parallel principles from classic operating system concepts, such as having fail-safe defaults. For instance, principle 2 says that rather than recording X, we should record everything *except* X, after having determined from principles 1, 3, and 4 that X is not valuable.

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

1. changes in the *user enviroment* (such as the UNIX shell or applications) (principles 1, 3, and 4)
2. changes in the *global system enviroment* (such as file permissions) that affects system operation at a later time (principles 1, 3, and 4)
3. *abstraction shortcuts* bypassing standard mechanisms, such as bypassing the filesystem to the raw disk (principles 1 and 4)
4. buffer and numeric *overflows* and *reading memory* locations larger than the allocated buffer (principles 1 and 4)
5. *race conditions* (principles 1, 2, and 4)
6. *programmer backdoors* [7] exploited (principles 1, 2, and 4)
7. *code injected* into the program instruction stream (principles 1, 2, and 4)
8. *code written to the heap* at runtime and executed dynamically (principles 1, 2, 3, and 4)
9. *self-modifying code* whose behavior varies based on environmental conditions (principles 1, 2, 3, and 4)
10. *illicit channels* [12], with high-volume data rates, such as data exposed from one user space to another with a memory write or data read from or written to raw disk at points with unallocated inodes (principles 1 and 4)
11. interception of *user input* (principles 1 and 2)

Throughout this paper, we will refer to a real system compromise as an example. A Mandrake Linux system was running a wide variety of security software, including *syslog*, *TCPWrappers*, the network IDS *snort*,[1] the host-level firewall *iptables*, and *Tripwire*.[2] All current security patches had been applied. Despite these typical precautions, the machine was compromised. This was discovered from email from a system administrator at another site, whose machines were being attacked by the compromised system. The vulnerability was probably in Linux-PAM (Linux Pluggable Authentication Module), though that suspicion is an inference from available and missing data. No hard evidence is available. The only evidence unearthed by standard procedures and tools was a directory containing a tool to perform brute-force `ssh` attempts against other machines, `ctime` evidence that a number of standard binaries had been replaced and possibly "trojaned," and syslog messages showing a number of successful `ssh` logins for every user on the system that did *not* have a login shell. No proof of how the intruder broke in and what the intruder did was found.

In the next section, we discuss the inadequacy of current forensic solutions in terms of our principles.

2. THE CURRENT PROBLEMS WITH FORENSICS

2.1 Principle 1: Consider the Entire System

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

[1]`http://www.snort.org`
[2]`http://www.tripwire.org`

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

For instance, current tools identify `mmap` or `sbrk` memory allocation system calls (syscalls), but do not tell us the size and content of data transfered to or from memory. Current tools identify a running process, but do not tell us which function in the code was exploited via buffer overflow to put the program in that state. Current tools can tell us which library was loaded and which file was executed, but not the directories in the user's library and execution paths that were searched, which would include information on how a user is attempting to explore the machine. Unlike many previous forensic tools and techniques [2, 8, 15, 16] or even fault tolerant operating systems, today's tools reduce the need for pre-determination of "attack" or "failure" methods by casting a broad net. But they omit non-kernel space data and information about user space that results in missing many forensically-relevant events.

Computer forensics typically uses a standard set of tools including system logs (*syslog* and *TCPWrappers* on UNIX, the *Event Log* on Windows), logs from network intrusion detection systems, host-level firewall logs, and *Tripwire*. None of these tools, either collectively or independently, are designed to consider the entire system. Despite their use, both initial knowledge of intrusions and methods of discovery are frequently based on guesses, as in our example above. In that example, we wanted to know, with high certainty, which software contained the vulnerability. What was the nature of that vulnerability? What backdoors were subsequently installed? What actions were taken on the system? What other systems were attacked from our system? How, and were any successful? No current tool is able to answer these questions well because no tools consider the entire system. Even the most basic forensic procedure of halting a machine to make images of the disks has a trade-off between preserving the contents of the disks, and wiping out the content of memory. Hence, all of these tools are inadequate for forensic analysis.

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

Two tools, *BSM* (Basic Security Module) [16] and *BackTracker* [9], gather enhanced data that is useful for forensic analysis, and both must be installed and running on a system *before* the events to be analyzed occur. The perspective gained from pre-installation is what gives them an advantage over other tools, though they are usually not used until it is too late, as was the case on the machine in our example intrusion. Despite their improvements over the standard tools, they still do not address the requirements for complete forensic logging: BSM's audit trails are too coarse and focused on the kernel space to capture understand events such as buffer overflow attempts, because BSM does not look at function calls and memory events. BackTracker helps answer the questions, "how did this get here?" and, "what effect did this have?" respectively. It can link processes, sockets, files, and filenames using syscalls and process executions, and output a visual, connected graph showing a chain of objects and events. However, these tools do not consider user space at all, and therefore will not uncover what caused not only our intrusion example, but the also the exploits we outlined in section 1.

One recent technique classifies forensic data necessary to design audit sources that have lower performance costs but continue to meet modern forensic needs [11]. However, this technique only looks at BSM data and data gathered from system library interposition. It ignores memory and user space. It does not address analysis (violating principle 5). It ignores a user bypassing dynamic system libraries by using their own static libraries (violating principle 2).

While UNIX *syslog* is at one extreme of the available logging solutions, *ReVirt* [4] is at the other. It provides an exact, replayable record of all *non-deterministic* events. This is, in some sense, the "ultimate" logging system. It records the least amount of information needed to re-create everything. But the recording of non-deterministic events is merely the first part of forensics, and is not analysis or auditing. ReVirt follows principle 1 by considering the whole system, but does not present events in a way that can be analyzed easily.

The only tool aimed at forensics that takes a broader view of the entire system is *Plato* [10], which uses *virtual machine introspection*. Hooks in Plato can monitor raw device I/O, CPU exceptions, kernel backdoors, syscall race conditions, file system race conditions, and virtual registers, RAM, and hard disks. At this time, Plato does not record memory events, and program-specific events and environment information, but could be expanded to do so. However, Plato does not record memory events, and program-specific events and environment information. Fortunately, an exact, complete re-creation of the operating system is not required to obtain a complete forensic view of the system. *There are many events that occur on an operating system that are irrelevant to forensics, and these events are easier to define and ignore than the events that* are *relevant to forensics.* Starting with this assumption would give us a more fail-safe method for having necessary information.

The hypervisor approach has several limitations. First, and most importantly, recording events by introspection upon replay may not give a precise replay of the original events. Introspection suffers from causing a "Heisenberg Uncertainty

Principle-like effect" of changing the data upon observing it. Precise timings are particularly vulnerable to interference. Also, hardware errors, even including bad sectors on a disk, which are common, could easily cause a different result upon replay. Second, virtual machines with "deterministic" replay are not always capable of replaying events exactly. Plato, ReVirt, and other hypervisor-style virtual machines with deterministic replay capabilities fail on multiprocessor machines, because neither the hypervisors nor the operating systems know the ordering of simultaneous reads and writes, by two or more threads running on different processors, to the same location in shared memory. The order of memory reads and writes is critical for deterministic replay. Thus, deterministic replay techniques are crippled on multiprocessor machines. The limitation of hypervisors with deterministic replay to uniprocessors is particularly critical given the prevalence of multiprocessor machines and the impending emergence of multiple processors on the same die sharing memory. It is critical to remember the precept that the original runtime provides the only authoritative record of what happened. Also important, using a virtual machine does not give access to the names of forensically valuable variables and functions within running programs. Finally, in many cases, there is limited practicality in running and maintaining virtual systems.

2.2 Principle 2: Assumptions Should Not Control What Is Logged

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

For example, ignoring *insiders*[3] to focus on *outsiders* places too much attention on access control mechanisms rather than recording system events. *Any user can be a threat.* One of our key desired outcomes is the ability to detect and prove when this occurs. In our intrusion example, all of the secure and encrypted access controls were not enough to stop or even log a remote exploit.

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

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

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

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

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

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

[3] "The *insider* is a user who may take action that would violate some set of rules in the security policy were the user not trusted. The insider is trusted to take the action only when appropriate, as determined by the insider's discretion." [3]

Though BackTracker can show a series of kernel-level objects and events leading to the creation of a particular file, it is merely showing a part of what happened, and not what *could* have happened at a particular stage, nor what vulnerabilities existed there. In our intrusion example, BackTracker may have shown that a shell was obtained following a Linux-PAM exploit, but what other actions could have been taken? Though BackTracker may help a forensic scientist decide where to look for a particular action or exploit, it does not necessarily help the analyst understand the nature of the action itself. For instance, process accounting may show evidence of a startup script running, and BackTracker may show a chain like `socket->httpd->sh`. This indicates that a shell was obtained from `httpd`, but does not help an analyst understand the nature of the vulnerability that allowed the shell to start. One way to analyze this is to ask the question at each stage: "What *could* happen here?" This approach allows an analyst to consider not only what is shown but also what else could have been done. Much like playing chess against a computer system and being able to see the computer's legal moves, this presentation should include a list of the intruder's actions, the *results* of those actions, and a tree showing other possible actions at each stage. Such a technique could assist not only with forensic analysis, but also *vulnerability analysis.* We want to know not only the events that transpired, but also the context and conditions set up by those events, and the capabilities achieved by the user in each condition [17]. In our intrusion example, we want to know not only the exploited program, but also the nature of that exploit and what else might have been vulnerable.

2.4 Principle 4: Context Assists in Understanding Meaning

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

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

2.5 Principle 5: Every Action and Result Must Be Processed and Presented in an Understandable Way

In general, forensic tools are not designed to do analysis. None of the standard tools installed on the machine in our intrusion example do any analysis. Even the "enhanced" tools, which require being installed ahead of time and are supposed to aid a human to do analysis, do so poorly or in passing. Most tools collect or display information so that humans can attempt to perform analysis. But the tools do little to analyze it themselves. The best tools only display collected information, and do not analyze, limiting their ability to present information in a coherent way. For instance, we can record keystrokes but we do not necessarily know the result of their entry [18]. We can record and view a chain of processes, but we do not know what took place within them at the level of memory accesses, and we do not know the results of other operations that depend on the current state of the user space environment. We need to analyze and correlate recorded events using enhanced logging techniques, about both kernel and user space events and environments, enabling an analyst to distinguish meaningful results from the actions that caused them and the conditions which permit them.

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

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

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

In our intrusion example, the standard tools did not present evidence of the intrusion in a coherent way. The evidence was scattered throughout syslogs and the filesystem in an unorganized and uncorrelated manner.

The standard tools for forensics fail to address all five fundamental principles of complete forensics. They do not address the entire operating system, they assume that user actions will generate system log events, keystroke evidence, or kernel event evidence; and they all completely fail to understand results as opposed to merely actions. In the next section, we suggest possible solutions.

3. NEW APPROACHES FOR A SOLUTION

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

3.1 Principles-Based Logging

This section discusses techniques that follow from the principles defined in the introduction, which address the problems with forensic analysis. For instance, principle 1 states that everything needs to be recorded, but principle 5 cautions us to do so in a way that first permits computer analysis, ultimately enabling more intelligent analysis by a human.

We will begin outlining principles-based tools by using selected techniques from existing tools. Principles-based tools must record the nature and timing of interrupts and traps to the kernel (including syscalls and their parameters) as well as output from the kernel and information about asynchronous syscalls that have been interrupted by an interrupt and restarted. Tools must record memory allocations in both the stack and the heap, including their origins and timings. They must also record events involving other standard interfaces, such as the filesystem and network stack, including opening and closing of file handles, disk reads and writes, packets sent, DNS queries, and their precise timings. But this covers only a subset of all possible events. We must address the forensic principles, especially and ultimately principle 5.

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

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

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

From an implementation perspective, several methods can capture information about events that occur in user space, and choice of one is primarily important insofar as it satisfies auditing demands 2. One is *introspection* or monitoring of a virtual machine. This technique has been used successfully [6, 10] with security and allows the host operating system to monitor everything that occurs in the virtual operating system. A key benefit over other solutions may be a relatively low performance overhead. Unfortunately, introspection of a running virtual machine for the types of events that are of forensic interest is likely to increase performance overhead significantly. Likewise, as mentioned earlier, introspection upon replay suffers from the problem of interfering with the events being replayed, or replaying imprecisely if hardware conditions changed.

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

NetBSD 2.0 contains a feature called *verified exec*[4] that can be used to impose restrictions on running only cryptographically-signed ("fingerprinted") binaries. In this way, having forensically valuable information compiled into binaries could be enforced. In the compiler-instrumentation approach, user space information can be recorded by instrumenting the system's C/C++ compiler and mandating that any binary run on the system, including the kernel, be compiled with the special compiler. This approach also has the benefit of saving more user-understandable information than virtual machines or binary rewriters because it can force all binaries to have debugging and profiling information compiled in. With binary rewriting, a binary can be instrumented to gather information similar to that which a compiler can give. The implementation is simpler, but the presence of the symbol table cannot be guaranteed.

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

Using compilers and binary rewriters raises the following problem: capturing information about the entire system requires that even the operating system be recompiled. If imposed on the kernel and drivers, this restriction could cause problems for code dependent on specific timing responses from the hardware. While not all systems have such dependencies, we would like our techniques to be generic. Fortunately, the parts of the code that rely on timing information interact with the hardware and need not be instrumented at the same level as all the other system and user code. Because we know the inputs to the kernel (syscalls, traps, interrupts), and the kernel is deterministic, we can determine its results via replay. Similarly, by using forensic data gathered from other programs and by drawing upon collected user space information (for instance, a hash of the memory image of the kernel [6]), we can determine how a kernel has changed and how that change has affected the system, without instrumenting the kernel itself.

[4]`http://www.netbsd.org/Releases/formal-2.0/NetBSD-2.0.html`

To date, we have made a conscious decision to concentrate on completeness and efficacy rather than efficiency and performance. We are well aware that there are significant performance considerations with the techniques that we suggest, however. Obvious approaches include information compression, co-processor-assisted logging [14], and dedicated hardware [19] for logging non-deterministic events.

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

3.2 Principles-Based Auditing

Though only represented by a single principle, the most difficult part of forensic analysis is auditing the data. A solution requires a method of presenting kernel and user space context and events together to the analyst. New tools need not exhaustively *analyzing* the data themselves, since this would require foreknowledge about the nature of the event, which is an assertion that we avoid (principle 2). Rather, a principles-based tool should exhaustively *log* data, and, in presenting it, enable the human analyst to perform analysis more easily and completely. A presentation should include the raw events themselves, and enable the ability to easily view events and environments at arbitrary points in time. It should also allow correlation of those events arbitrarily with others at similar points in time or operating on similar points in memory or on disk. An analyst should be able to easily speculate about global questions involving forensic data ("were there any potential memory race conditions recorded in this day-long time period, and where?") and also to look in more detail [8] into the macro-views of process and file information provided by existing tools to find answers.

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

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

Once translations and correlations are finished, principles-based tools can perform automated analyses to generate warnings for a human analyst. As with intrusion detection, automated methods can be both anomaly-based and signature-based. Anomaly detection is always an available tool, but is useless when the "attack" in question is "normal," common, or innocuous enough not to appear as an invariant [5]. However, if modeled correctly, even those can be discovered through signature detection. Usefulness of anomaly detection and signature detection to forensics may be exactly the inverse as their uses to intrusion detection, because in forensics, we do not have to predict signatures in advance, and can refine them after the fact.

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

Covert channels are difficult to eliminate, and even where they are preventable, often it is undesirable to do so. On the other hand, *storage channels* and *legitimate channels* [12], are currently as badly audited in today's operating systems as covert channels. A principles-based forensic system would possess the necessary data, since it logs all such data.

A final method of analysis required for principles-driven tools, addressing principle 2, is to perform analysis not only about what *did* happen, but what *could* have happened at each step in a system's execution, both in kernel and user space. For example, in a buffer overflow, the return address is typically altered to return the execution point to an alternate location. In this automated analysis, we also want to know the other active programs and their functions that could have been jumped to. Or, in a possible race condition situation, we want to know the programs and nature of the objects involved. To perform this analysis, one might use requires/provides techniques [17] as a model to present abstract events, and look not just at that series of events but also at a set of *conditions* and *capabilities* acquired given the events and the context in which they occur.

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

3.3 Summary of Principles-Based Solutions

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

4. CONCLUSIONS

The principles of computer forensics we have described help us devise techniques to significantly improve our ability to understand what has happened previously on a computer system, when compared with current tools. Those techniques, which we have also outlined, do not require predetermination of the nature of the events or the skill level of the attacker, and do not require the analysis to begin with knowledge of precise details after the fact about users, times, processes, and system objects involved. The techniques also have the potential to perform their work in a practical way.

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

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

5. ACKNOWLEDGEMENTS

This material is based on work sponsored by the United States Air Force and supported by the Air Force Research Laboratory under Contract F30602-03-C-0075. Matt Bishop gratefully acknowledges support from awards CCR-0311671 and CCR-0311723 from the National Science Foundation to the University of California at Davis. Thank you very much to Roger Bohn, Larry Carter, Drew Gross, and the NSPW 2005 delegates whose comments greatly improved this work.

6. REFERENCES

[1] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, April, 1980.

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

[3] M. Bishop. The Insider Problem Revisited. In *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, 2005.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, 2002.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Symposium on Operating Systems Principles*, 2001.

[6] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium*, 2003.

[7] D. P. Gilliam, T. L. Wolfe, J. S. Sherif, and M. Bishop. Software security checklist for the software life cycle. In *Proceedings of the Twelfth IEEE International Workshop on Enabling Technologies: Infrastructure for Colaborative Enterprises (WETICE'03)*, 2003.

[8] A. H. Gross. *Analyzing Computer Intrusions.* PhD thesis, University of California, San Diego, 1997.

[9] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, February 2005.

[10] S. T. King, G. W. Dunlap, and P. M. Chen. Plato: A platform for virtual machine services. Technical Report CSE-TR-498-04, University of Michigan, 2004.

[11] B. A. Kuperman. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources.* PhD thesis, Purdue University, 2004.

[12] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[13] S. Peisert. Forensics for System Administrators. *;login:*, August 2005.

[14] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 2005 USENIX Security Symposium*, 2005.

[15] T. Stallard and K. Levitt. Automated analysis for digital forensic science: Semantic integrity checking. In *Proceedings of the 19th Annual Computer Security Applications Conference*, December 8-12 2003.

[16] Sun Microsystems, Inc. *Auditing in the Solaris^{TM} Operating Environment*, February 2001.

[17] S. J. Templeton and K. Levitt. A Requires/Provides model for computer attacks. In *Proceedings of the 2000 New Security Paradigms Workshop*, pages 31–38, 2000.

[18] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[19] M. Xu, R. Bodik, and M. D. Hill. A “Flight Data Recorder” for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, 2003.