Using the Post-Compromise Audit Trail: Paths of Communication

Cory Williams, Advisor: Greg Ganger

May 3, 2002

1 Abstract

Today, computer logging systems are configured with the assumption that they will not continue to function after a system intrusion. This paper discusses what information such systems might log so that audit logs collected during an intrusion might serve to better diagnose the intrusion. This requires some logging mechanism that the intruder cannot tamper with. While this paper does not propose any such mechanism, we discuss possibilities to help determine what should be logged.

A system of logging kernel events is considered with the intent of reconstructing what was happening in the system during the intrusion. We consider things on the granularity of processes and the paths of communication between them. This is used to trace possible cause and effect in the system. An implementation of this logging system is presented and measured to demonstrate feasibility in terms of bandwidth and performance impact.

2 Introduction

Computer compromises are by no means a new thing. Numerous break-ins, both high profile and unpublicized, occurred throughout the 1990s. Many of the basic methods of security compromise have stayed the same; The buffer overflow technique used in the 1988 Morris worm is still in use today. Security flaws and system compromises are a fact of life.

When these compromises are discovered, computer forensics specialists can be called in to examine the system and determine what happened. Some of the most useful information in such an analysis is system logs. These logs are records of events that take place in the system and succinctly let us know what is happening within the computer. When a significant event occurs in the operating system or an application, a message is generated that describes the event and any significant details. These messages are recorded for a number of reasons, one of which is for security auditing.

Measures must be taken before hand to ensure the integrity of these system logs during a compromise. Two methods that accomplish this are adding cryptographic hashes to the log files and immediately sending the log messages over the network to a remote computer dedicated to storing these logs. This allows the verification of the integrity of the logs up until the point when the intruder has taken over the system. At that point, the intruder may have control of the logging mechanism and thus the messages produced cannot be trusted. The log messages recorded right before this occurs can be very helpful in identifying how the intruder got into the system.

The logs do not help determine what the intruder did while in the system. For this we would need accurate logs after the intruder compromised the system. The solution to this problem today is to take a disk image as soon as possible after the compromise and to carefully examine it. Deleted files and other telltale marks left behind can help an expert determine what happened. In the end, the contents of the disk are usually declared a complete loss and a full restore from backup must be done.

What if we did have accurate logs after the point at which the intruder compromised the system? With the right information, all actions of the intruder in the compromised system could be reconstructed. With such information in hand, a partial restore could be done instead of a full restore, the intent of the intruder could be better understood, and the how and when of the initial intrusion could be more easily deduced. This may help with intrusion detection as well, since overtly unusual actions (e.g. deleting important files) can be detected.

This paper explores the value of logging the existence of all pathways of communication in a computer, both between processes and with the outside world. This information can be use to trace paths of possible influence within a system. From this, we can determine what might have caused a particular event, as well as everything that may have been affected by some initial input into the system.

3 Design Issues

To effectively decide what to log, we must first decide which questions we want answered by the log data. There are two major questions that are directly addressable by data from the logs: "How and when did the intrusion occur?" and "What was affected by the intrusion?" These can be broken down into more detailed questions that point directly to what we need to log.

We must also temper our decisions with considerations of what is feasible to log. First, we want the generation of the messages to have a low impact on system performance, including processor overhead and bandwidth to disk or the network. Second, we must consider the mechanism that performs the logging and what limitations that might place on what can be logged.

The latter point is a difficult one and this paper does not try to propose any such mechanism. Rather, we will consider the possibilities for such a mechanism and try to choose our log messages in light of these.

3.1 The Questions

The ultimate purpose of any logging system is to answer questions. In the case of intrusions, there are two major categories of questions. First, we can ask how and when the intrusion itself happened. Answers to this are central to fixing the security hole and bounding the intrusion in time. Second, we would like to know what the intruder did to the system after the compromise. Answers to this question can help us restore the system to its preintrusion state and determine what information or other systems might have been compromised.

The matter of how and when the intrusion occurred can be broken down into many subquestions:

- At what time did the intrusion happen?
- If a process was exploited, what process was it?
- If it is a local exploit, what was the original user id?
- Did the intruder gain access through improper configuration?

The matter of what the intruder did while in the system can also be broken down in many subquestions:

- What files did the intruder access (read and write)?
- Did the intruder connect to any other computers?
- Did the intruder alter any processes? (to capture passwords, etc.)
- Did the intruder copy in any tools or scripts?
- Did the intruder reconfigure any hardware/firmware?

These questions are closely related to one another. There are other equally valid questions to be asked of a compromised computer system, but these are a representative sample. The messages we log must be chosen so that these questions can be answered.

3.2 Logging Mechanism

Ideally, logging would be an integrated feature of every process running. Each process could record high-level messages detailing exactly what is happening. Unfortunately, we cannot expect the intruder to leave such features in place and we can be quite certain that the intruder's tools will not contain such features. The logging mechanism must be a small, self-contained mechanism capable of operating independently from every other process on the system.

Two possibilities come to mind. The first is code within the kernel of the operating system. In virtually every operating system, the security of the whole system is based on the security of the kernel. If any process can alter the kernel in arbitrary ways, there is no hope of keeping any semblance of security and access control. This makes the kernel the single point of failure in a system and thus the most important code to keep secure. If any code has a hope of staying secure in the face of an intrusion, the kernel does. This (hopefully) has the added effect that the most effort is put into making the kernel secure.

Adding an effective logging mechanism to most kernels would require some modification to the model of access.

In most systems today, the superuser has the power to change virtually anything in the system. Further, access to the superuser privileges is generally protected only by software. For this logging mechanism to be effective, it should be hard to change any settings related to logging, at least without detection. Additional hardware such as a physical key for the machine is a possibility. Requiring terminal access for changes is another. There could also be a mechanism that always pages the sysadmin when a change in the logging system configuration occurs, something that should not happen without his knowledge. These are tall orders. In general, this is not an idea intended for the average desktop machine, but rather well administered machines running without the need for constant changes.

A second possibility for the logging mechanism is specialized hardware. Even though such hardware does not exist today, such hardware would probably have to be as simple and general as possible. It is unlikely that hardware would be designed to keep track of high level events. Rather it would be involved with the lowest-level concepts, like separate address spaces for separate processes. Such hardware has precedent in the MMU. Most modern systems have hardware that assists the operating system in implementing virtual memory. Such hardware provides the most general possible services so that it can be used by virtually any operating system. Hopefully research such as this will spur and help guide the development of such hardware. Research has been done recently on adding security features to hardware devices, though not specifically the types of hardware needed for secure OS event logging [GN01].

Both of these possibilities lead us to examine the lowest levels of the system. Since the hardware in question is hypothetical, we will proceed with things on the level of the kernel, and we will try to keep the concepts general and OS agnostic.

4 Logging I/O

One of the basic units within any operating system is the process. Processes are protected from one another and cannot access each other's memory. In fact, isolated processes cannot communicate with each other in any way and thus one cannot affect the eventual outcome of another's computation. To pass information between them, I/O channels must be set up and used. The mechanisms used vary from simple signals and message queues, to more complicated schemes, like file systems. Processes can also set up communication with the outside world through devices like keyboards and network cards.

Any person wishing to use a given computer system must send some information to some process on that system. That process may interpret that information and pass its own information on to other processes, causing them to act in the intended manner. If there is no path for information to pass from a person to some process, then there can be no influencing the actions of that process.

These I/O channels form a dynamic network in the computer. Analysis of this network of communication can give us all possible paths that information could have taken and thus all possible paths of influence in the system. Using this information we can make judgments regarding the possibility of one event in a system affecting another event.

All of these I/O channels are managed by the kernel or some other core part of the OS. These are ideal candidates for the messages we wish to be logging. They are low level and observable by the kernel. As we will show, they are sufficient to answer the questions posed earlier. We can also keep the overhead of logging such messages to a minimum if we log only the existence of the connection, and not the information passed over the connection.

We must also consider covert channels. It is not possible to log the existance of such channels by their very nature. This is not a major problem when performing analysis of influence within a system. A system running in a non-compromised state should not be using any covert channels for communication. An intruder must use the existing channels to perform the break-in, which we can track. Once inside, he may use covert channels, but he must set them up himself and can only use them between processes that he has influenced already to set up the channel. Thus the covert channel will not influence anything that has not been influenced otherwise.

4.1 Types of I/O

There are many types of I/O subsystems in a modern OS. The simplest can be considered together while the more complicated systems require special attention.

IPC, Shared Memory, Pipe, Connected Sockets These and similar methods of communication have explicit connection semantics. There is a small number of system calls which control these I/O channels and they are easy to track. Furthermore, the processes cannot set them up alone and must involve the kernel. Most have only two connected processes, further simplifying the information needed to be recorded. The only caveat of note is that we should also check if the channel is actually used. One end point of a channel may be passed to another process before it is used.

Unconnected Sockets Protocols like UDP have no notion of an established connection. Instead, each packet of information can be sent to a different address. This must be handled carefully so as to not create too high a rate of log messages. One solution is to log only the first that a socket sends a packet to a particular address.

Direct Process Access By this we refer to attaching to a process to debug it, or otherwise directly accessing the memory of another process. This can be seen as another form of bidirectional communication.

Creating new processes When a process forks, both the new and the old processes continue to carry out the intention of the old process. Even when a new binary image is loaded to run, the parent is choosing this new image and the arguments passed to it. Thus in both cases, we consider the parent to have communicated its intent to the child.

File System This is the most complicated mechanism of communication. In general, if one process writes to a file and then a second reads from the same file, we can consider the first process to have communicated with the second. This then allows more than two processes to communicate with each other through a file.

There are a number of complications. First is manipulation of the naming layer. This must also be tracked and if one process alters the namespace and then a later process uses that part of the namespace, we must consider communication or manipulation possible. This is the mechanism used for race conditions, i.e. symlinks are changed in between uses of them.

Second is the use of multiple names for the same file. Hard links allow two names for a file with neither containing any information about the other. The solution to this is that we must always record some identifier of the file along with the name, e.g. the inode number.

Last, there is a question of granularity. The simplest choice is to log only that a file was accessed. The most complicated choice is to log which bytes were read or written. The former choice keeps the rate of messages low and analysis easier. The latter choice is necessary in some cases, though, like databases. The windows registry is a good example of such a file. Almost every process will access it for both read and write, though in very different locations. The most effective systems would combine these two, logging byte ranges only for chosen files.

These considerations cover the features present in common file systems like ext2 and ntfs. Other file systems may have further complications.

4.2 Problems

There is a major problem with the way modern computer systems are set up. Fortunately, it is not an insurmountable problem. The issue is one of multiple interfaces to the same resource. One example is access to the disk through the file system and through a raw, block interface. If the intruder is allowed to write arbitrary blocks of the disk, he can alter a file and there would be no log message linking that write to the file it altered.

One solution to this is to do just what the file system itself does. If someone writes to the raw interface, the kernel would translate this up to the level of the file system and determine what it was affecting. In some cases it would be affecting metadata and we would have further problems of deciding how that affects what programs see. We would have to try and interpret the bytes written and see how the naming system was affected. While not an impossible task, this would be extremely complicated and time consuming.

The second solution is to just completely disallow access to the raw interfaces, except for the privilege level that can alter the logging system. This makes the problem go away, although it could be a issue if some process needed access to the raw interface. That process alone could be given access, and if it is compromised, then we give up on trying to interpret the paths of communication through the file system.

5 Other things to log

The previous section describes only one category of information that should be logged. There are other categories of things to log which help with a post-compromise analysis. These are more common sense things; two such categories are capability/access right changes and any significant changes in the kernel state. While this is not the focus of this paper, it is important to consider such information in the context of this paper.

Every process in a system has a set of capabilities that change over time. Any such changes should be logged so that later the complete set of capabilities can be reconstructed for any point in the lifetime of that process. This can be easily performed by recording or knowing the initial state of a process and recording any changes made. Since virtually all compromises involve the intruder obtaining privileges that were not intended for him, a record of this privilege elevation can be very useful.

The kernel and the happenings inside of it stand outside the realm of processes and I/O channels. If there is internal state the kernel maintains that can affect the system, it should be recorded.

6 Answering the Questions

Once we have recorded the needed information and reconstructed the paths of communication within a system during a period of time, there are two possible ways to analyze the data. The first is to construct a process tree of all processes that ran in the system and then augment that tree with the other paths of communication. This diagram could be displayed visually and used by a system administrator or computer forensics expert to examine what was happening in the system. Such a diagram would contain quite a bit of information, and the principle problem would be finding ways to display it effectively.

The second way to analyze the data is to collapse groups of mutually communicating programs down into a single data point. Processes that have a one-way communication between them would not be collapsed. This will give us a directed graph which presents the flow of information through the system. These paths of communication can be used to determine what could have possibly influenced what. Such a graph would be much simpler to examine than the aforementioned process tree.

This second representation leads us to consider what automated analysis we could perform. Consider that to alter the behavior of some program, an intruder must communicate with that program or in some other way alter its internal state. This would show up in our diagrams. Given a particular action which we know the intruder has performed (e.g. obvious actions such as deleting important files), we can automatically examine the graph of communication and find all possible outside connections which could have lead to that action. This will give someone a starting point of how and where the intruder initially entered the system.

Analysis of the privilege levels of processes can also be fruitful. Consider an intrusion that made use of a setuid exploit. Given a known point of intrusion, the paths of communication can be traced backwards to find all places where the privilege level of a process was elevated. This should produce a relatively short list that can be examined by hand for unusual activity. Intrusion detection could also be applied to try and find such processes that act in an unusual manner.

Perhaps more importantly, once the point of intrusion has been identified, the graph of communications can be examined to determine all possible files, processes, etc. that could have been affected by the intruder. This is an upper bound for the damage done by the intruder and allows for a more selective recovery. Untouched files can be carried forward and only the suspect files have to be restored from backup. We can also see if the intruder tried to contact any other computers or may have intercepted any sensitive information. This can aid in determining what other computers or accounts may have been compromised.

Given such a list of initial entry points or what might have been affected by the intruder, very select portions of the full augmented process tree can be examined for unusual activity.

7 Implementation

To demonstrate the feasibility of such a logging system, a prototype was developed for Linux. This prototype contains no security features, but rather focuses on demonstrating that the data stream of log messages does not consume excessive bandwidth or CPU cycles. Red Hat 7.2 was used with a 2.4.7 kernel. The kernel itself was altered to record log messages and a user program was written to interpret these events and perform analysis on them.

All relevant messages from the kernel were recorded, though the subsequent analysis focused only on certain types of I/O channels, specifically the file system and the process tree. One analysis consisted of specifying a processes that was recorded and tracing the paths of communication to determine all other processes that may have influenced the behavior of the original process. Another analysis consisted of finding all processes that may have been influenced by a given process. The program was also capable of listing all relevant recorded facts about a single process such as all files accessed, or all children forked.

7.1 The Kernel

The kernel was patched in several ways. The system call entry vector was altered to record select system calls and their parameters. In addition, during a call to `execve`, the close-on-exec flags were recorded to correctly update the file descriptor tables during reinterpretation. System calls recorded include all those involved in creation and manipulation of I/O channels and those that alter the privilege level of the process (e.g. `setuid`).

In the file system, the name resolution routine `path_walk` was instrumented to record all the inodes that are examined in the process of looking up a filename. Several other functions in the file system were similarly instrumented to record newly created inodes. For each inode, only the inode number and device number where recorded. It is assumed that this information can uniquely identify any file.

See Appendix A for a full list of events that were recorded.

All this information was written directly into a file residing on the local hard disk for later retrieval.

7.2 Reinterpreting Log Messages

After a set of log messages have been recorded, a program goes back through these messages and reconstructs the actions of the processes. The basic idea is to reconstruct the relevant state that the kernel maintained during the generation of the log messages. A timeline of events for each process is created and can be used to determine which processes were in communication and which channels they used.

This gives us a graph-like structure of communication within the system. Given any point in any process, we can easily work forwards or backwards in this structure to determine what

Run	Mean lat. (secs)	Mean GETS/sec	log size (MB)	gzipped log (MB)
clean run 1	0.3127	88.98	-	-
clean run 2	0.3064	89.32	-	-
clean run 3	0.3219	88.69	-	-
logIO run 1	0.3056	89.37	26.10	2.43
logIO run 2	0.3070	89.32	26.12	2.43
logIO run 3	0.3066	89.34	26.15	2.44
logRW run 1	0.3057	89.21	32.26	3.58
logRW run 2	0.3059	89.22	32.17	3.58
logRW run 3	0.3039	89.22	32.25	3.58

Table 1: Results of Surge web server benchmark

other processes could be influenced from this point, or which processes could have influenced this point.

8 Measurements and Results

Several methods were used to measure the impact of this logging system. The Surge web server benchmark [Cro], the HBench-OS benchmarks [Bro], and a full compilation of the Linux 2.4.7 kernel were used. The performance of these were measured using a normal kernel and the custom kernel that performs the logging of the relevant messages. For the custom kernel, two configurations were used. In both, all events altering the privilege level of a process are recorded, as well as all events concerning the creation, subscription, or deletion of I/O channels. In the second configuration, the length of all reads, writes, and seeks were recorded so that we could determine specifically which byte ranges of a file were accessed. We will refer to these configurations as clean, logIO, and logRW. It should be noted that all processes on the system were logged, but there was nothing else actively running on the machines during the time the benchmarks were run.

In addition the analysis program was informally evaluated to test its correctness.

8.1 Surge Web server Benchmark

The Surge benchmark was run for 500 seconds on each of the three configurations three times. It was configured with the default parameters given in the accompanying HOW-TO document. A total of 200 client threads where used to make request to the server and this was found to saturate the server. Apache with the default configuration was used for the web server. Table 1 summarizes the results.

In both custom configurations we see that the logging of messages does not noticeably impact the performance of the web server. Observe that log files were reduced considerably in size when gzipped. This points to much redundancy in the log messages. Some of this arises from actual redundancy in the information needed to be recorded and some arises from inefficiencies in the way we have recorded the messages. A better designed system

Benchmark	without logging (μs)	with logging (μs)
Time to execute connect()	170.50	183.83
Time to create empty file	1498.41	1678.01
Time to execute fork()	500.15	508.96
Time to exec() hello-world	12630.47	12987.10
Time to execute write() (zero length buffer)	1.32	2.322

Table 2: Results of the HBench-OS benchmarks on system call latency

could easily improve on the current data rate of log messages.

The data rate for the log messages is relatively low for this benchmark. In the case of logIO, we have a data rate just under 5 KB/sec (using the gzipped log file). For logRW, the data rate is about 7.3 KB/sec. Note that the read and write messages account for almost a third of the logRW messages.

8.2 HBench-OS Benchmarks

HBench-OS is a suite of benchmarks that measure basic system performance. Specifically we are interested in the latency of system calls. The clean and logRW configuration were benchmarked. The log messages were sent to `/dev/null` for these benchmarks since they are written out in large blocks and we would not want that write to penalize the syscall that fills the internal buffer and skew the results. The results are summarized in Table 2.

The results for this benchmark show the largest difference in performance. This is to be expected, though, since the performance penalty is in the additional code used to do the logging inside the system calls. Note that the creation of a file and `exec` both have larger penalties than the others. This is because we must record all inodes examined in the path lookup, generating more messages than just a simple noting of the system call.

8.3 Kernel Compilation

The compilation of a Linux kernel was chosen to measure the performance impact of the logging system because it creates many processes to compile each file and performs substantial file I/O. The 2.4.7 Linux kernel was used and the benchmarks includes generation of the configuration files, dependences, and the kernel itself. The kernel was compiled three time with each of the configurations. The results are summarized in Table 3.

In this benchmark we can see a small difference when the logging is turned on. Turning the basic logging on adds about 16 seconds and adding the read/write logging adds about two more seconds. This is only about a 1% increase in time over the clean run. The data rate of the log message, 8.2 KB/sec for logIO and 10.2 KB/sec for logRW, is greater than for the Surge benchmark, but still within a reasonable range.

Run	Time (min:sec)	log size (MB)	gzipped log (MB)
clean run 1	22:09	-	-
clean run 2	22:01	-	-
clean run 3	22:08	-	-
logIO run 1	22:26	159.02	11.06
logIO run 2	22:25	159.02	11.06
logIO run 3	22:26	159.03	11.06
logRW run 1	22:26	175.21	13.32
logRW run 2	22:29	175.21	13.32
logRW run 3	22:27	175.21	13.32

Table 3: Results of the Linux kernel compilation

8.4 Analysis Program

The analysis program was tested by performing some number of simple actions chosen to use a variety of paths of communication through the file system and the process tree. The resulting log file was then analyzed to determine if those actions were properly reconstructed. The communication paths used were:

- Writing to a file and subsequently reading it
- Writing a binary file and subsequently running it
- Changing access rights on a file and subsequently listing its parent directory
- Moving an entire directory and subsequently reading a file contained in it
- Creating a symlink to a directory and reading a file with that symlink in the path
- A process forking a child

These were tested individually and in several combinations. An example of such a combination is process A creates a symlink which process B then uses to open a file for writing. Process B then writes a binary file which is run as process C. Process C copies a file and process D then reads the copy. If the analysis program works correctly, it should say that process A (as well as the process that forked it) affected process D. In fact, the analysis program correctly reported the expected causal dependences in every case.

9 Conclusions

Both configurations of the logging system do not have an undue impact on the performance of the system. The performance as measured in the Surge benchmark and kernel compilation show us that the logging system impacted the total run time only slightly. While the data rate of the log messages is not insignificant, even 10 KB/sec is only a small fraction of the bandwidth of a modern disk or 100Mb/sec Ethernet.

This data stream must be stored if we are use it later. If we consider the 10 KB/sec stream, we must record 843 MB/day. With a 10 GB disk, we could store approximately two weeks worth of data. This is not an insignificant investment. Thus this is not a solution for every desktop, but rather one for large servers that are prone to attacks.

It is not unthinkable to use this logging system on a well administered desktop machine. We don't expect the system to be fully loaded all the time. For such a system, we could estimate half the data rate of the Surge benchmark without recording of reads and writes. At 2.5 KB/sec we must record 211 MB/day meaning we could keep 2 weeks of logs in about 3 GB of space. Given that a 30 GB hard disk is a common size for a desktop today, 3 GB is not completely unreasonable.

It is also encouraging that this system was designed as a quick prototype and not as a fully tweaked system. There are numerous improvements that could be made to decrease processor time and the message data rate.

The testing of the analysis program was not comprehensive, but it was performed largely as a check to make sure that all needed information was being recorded correctly. There may still be unseen bugs or things that we have forgotten to analyze or test, but we would hope that these errors are small in number and that we are logging at least a majority of the information really needed.

Overall, these results give promise for further research. This paper has explored only a fraction of the possibilities. Much work needs to be done on find efficient analysis of these results and effective ways of presenting the data to be analyzed by a person. Work must also be done on designing a robust logging system that will make these ideas feasible in light of real intrusions.

10 Future Research

The logging system as described and implemented in this paper could use many improvements. Currently, the log messages are collected from a single point (system call entry) which eased implementation. But a more integrated logging system would help lower the data rate of log messages as well as the CPU load. Ideally, every change of state in the kernel could be recorded at the point that change is made instead of from the indirect level of system calls.

As the results show, the current system does not do the best job of recording only the information needed. There is redundancy in the data that perhaps could be better eliminated by doing some amount of preprocessing on the log messages. One good example is that instead of logging the length of each read and the position of each seek, a set of ranges could be built up in the kernel for each file that would be written out periodically. A sequential read of a large file would show up as a single message instead of many 4KB reads.

The logging system currently ignores the preexisting state when it is started. This would certainly need to be fixed before this system can be considered actually useful. The simplest solution would be to record this existing state when the logging system is first stated or always have the logging system running. This is another argument for deeper integration with the kernel.

Another idea would be to integrate intrusion detection with this sort of logging system. Intrusion detection would benefit from being able to examine the system after it is compromised. Complex intrusion detection, such that it cannot be run realtime, could also be applied after the fact, when time is plentiful. One problem facing all intrusion detection systems is that of false positives. Many intrusion detection systems generate a likelihood of the system being compromised at any given point. But the decision of whether or not to flag a system as compromised is a binary one, so some cutoff for the likelihood must be used. This leads to a problem of false positives that must be dealt with by intrusion detection systems [Axe98]. During a post-compromise analysis, it does not have to be a binary decision. Likelihood of break-in can be attributed to all points in a process's lifetime and this measure can be used to determine a ranked list of likely compromises. A particularly good intrusion detection system for this would be [FHSL96] since it also examines system calls.

One major weaknesses of the current system is a lack of discrimination. It can eliminate some parts of the system when determining what caused some event or what some event could have caused, but this still leaves numerous events to examine. Many of these actually have no real connection to the event in question. More research needs to be done to further narrow down what events should be examined.

11 Acknowledgements

I'd like to first thank my advisor Greg Ganger who has provided me with inspiration, encouragement, support, and good advice. I'd like to thank John Strunk for his ideas and help all along the way and Jay Wylie for his comments on this paper. I'd also like to thank the Parallel Data Lab for the use of its facilities and equipment.

I thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. I thank IBM and Intel for hardware grants supporting our research effots. This work is partially funded by the Air Force Office of Sponsored Research (Air Force grant number F49620-01-1-0433).

A Events Logged

Tables 4 through 11 list all the Linux system calls and which are recorded as part of the logging system implemented for this paper. Table 12 lists the calls that are multiplexed through the `sys_socketcall` system call. Table 13 lists the calls that are multiplexed through the `sys_ipc` system call. Table 14 lists the other events that are recorded from the kernel. In this last table, the function `path_walk` is the name lookup routine used by the Linux virtual file system.

References

[Axe98] Stephan Axelsson. Research in intrusion-detection systems: A survey. Technical Report 98–17, Department of Computer Engineering, Chalmers University of Technology, December 1998.

[Bro] Aaron B. Brown. Hbench-os suite of benchmarks. http://www.eecs.harvard.edu/ vino/perf/hbench/.

[Cro] Mark E. Crovella. Surge webserver benchmark. http://www.cs.bu.edu/faculty/crovella/links.html.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[GN01] Gregory R. Ganger and David F. Nagle. Better security via smarter devices. In *Proceedings of the 2001 IEEE Workshop on Hot Topics in Operating Systems*, pages 100–105. IEEE Computer Society Press, May 2001.

Table 4: List of system calls used in Linux

#	Name	Log	Notes
0	defunct setup() call	N	
1	sys_exit	Y	Return value could be useful for analysis
2	sys_fork	Y	Needed to trace process tree
3	sys_read	Y	Recorded only for logRW runs. Needed to determine byte ranges accessed. The buffer read is not recorded.
4	sys_write	Y	Recorded only for logRW runs. Needed to determine byte ranges accessed. The buffer written is not recorded.
5	sys_open	Y	Needed to trace file access
6	sys_close	Y	Needed to trace file access
7	sys_waitpid	N	Provides no information obtained otherwise
8	sys_creat	Y	Needed to trace file access
9	sys_link	Y	Needed to trace file access
10	sys_unlink	Y	Needed to trace file access
11	sys_execve	Y	Needed to trace process tree
12	sys_chdir	Y	Needed to trace file access
13	sys_time	N	Doesn't change system state
14	sys_mknod	Y	Needed to trace file access
15	sys_chmod	Y	Needed to trace file access
16	sys_lchown16	Y	Needed to trace file access
17	defunct break() call	N	
18	sys_stat	Y	Needed to trace file access
19	sys_lseek	Y	Recorded only for logRW runs. Needed to determine byte ranges accessed.
20	sys_getpid	N	Doesn't change system state
21	sys_mount	Y	Useful for analysis
22	sys_oldumount	Y	Useful for analysis
23	sys_setuid16	Y	Needed to trace process privilege level
24	sys_getuid16	N	Doesn't change system state
25	sys_stime	Y	Changing time is an unusual event that should be logged
26	sys_ptrace	Y	Needed to trace which processes is debugging which
27	sys_alarm	N	Doesn't affect anything outside calling process
28	sys_fstat	Y	Needed to trace file access

Table 5: List of system calls used in Linux, continued

#	Name	Log	Notes
29	sys_pause	N	Doesn't affect anything outside calling process
30	sys_utime	Y	Needed to trace file access
31	defunct stty() call	N	
32	defunct gtty() call	N	
33	sys_access	Y	Needed to trace file access
34	sys_nice	Y	Could be useful in analysis
35	defunct ftime() call		
36	sys_sync	N	Shouldn't affect the state of the system as seen by processes
37	sys_kill	Y	Needed to trace use of signals
38	sys_rename	Y	Needed to trace file access
39	sys_mkdir	Y	Needed to trace file access
40	sys_rmdir	Y	Needed to trace file access
41	sys_dup	Y	Needed to trace file access
42	sys_pipe	Y	Needed to trace use of pipes
43	sys_times	N	Doesn't change system state
44	defunct prof() call	N	
45	sys_brk	N	Doesn't affect anything outside calling process
46	sys_setgid16	Y	Needed to trace process privilege level
47	sys_getgid16	N	Doesn't change system state
48	sys_signal	N	Doesn't affect anything outside calling process
49	sys_geteuid16	N	Doesn't change system state
50	sys_getegid16	N	Doesn't change system state
51	sys_acct	Y	Needed to trace file access (process accounting information is written to a file)
52	sys_umount	Y	Useful for analysis
53	defunct lock() call	N	
54	sys_ioctl	Y	Needed to trace file access
55	sys_fcntl	Y	Needed to trace file access
56	defunct mpx() call	N	
57	sys_setpgid	Y	Needed to trace process privilege level
58	defunct ulimit() call	N	
59	sys_olduname	N	Doesn't change system state
60	sys_umask	Y	Not actually needed to trace file system, but could be useful in analysis
61	sys_chroot	Y	Needed to trace file access

Table 6: List of system calls used in Linux, continued

#	Name	Log	Notes
62	sys_ustat	N	Doesn't change system state
63	sys_dup2	Y	Needed to trace file access
64	sys_getppid	N	Doesn't change system state
65	sys_getpgrp	N	Doesn't change system state
66	sys_setsid	Y	Needed to trace process privilege level
67	sys_sigaction	N	Doesn't affect anything outside calling process
68	sys_sgetmask	N	Doesn't affect anything outside calling process
69	sys_ssetmask	N	Doesn't affect anything outside calling process
70	sys_setreuid16	Y	Needed to trace process privilege level
71	sys_setregid16	Y	Needed to trace process privilege level
72	sys_sigsuspend	N	Doesn't affect anything outside calling process
73	sys_sigpending	N	Doesn't affect anything outside calling process
74	sys_sethostname	Y	Useful in analysis
75	sys_setrlimit	N	Doesn't affect anything outside calling process
76	sys_old_getrlimit	N	Doesn't change system state
77	sys_getrusage	N	Doesn't change system state
78	sys_gettimeofday	N	Doesn't change system state
79	sys_settimeofday	Y	Useful in analysis. Similar to stime
80	sys_getgroups16	N	Doesn't change system state
81	sys_setgroups16	Y	Needed to trace process privilege level
82	old_select	N	Does not provide any information not received otherwise
83	sys_symlink	Y	Needed to trace file access
84	sys_lstat	Y	Needed to trace file access
85	sys_readlink	Y	Needed to trace file access
86	sys_uselib	Y	Needed to trace file access
87	sys_swapon	Y	Rare event that could be useful in analysis
88	sys_reboot	Y	Rare event that could be useful in analysis
89	old_readdir	Y	Needed to trace file access
90	old_mmap	N	Just changes access method for a channel, not its status

Table 7: List of system calls used in Linux, continued

#	Name	Log	Notes
91	sys_munmap	N	Just changes access method for a channel, not its status
92	sys_truncate	Y	Needed to trace file access
93	sys_ftruncate	Y	Needed to trace file access
94	sys_fchmod	Y	Needed to trace file access
95	sys_fchown16	Y	Needed to trace file access
96	sys_getpriority	N	Doesn't change system state
97	sys_setpriority	Y	May be useful for analysis
98	defunct profil() call		
99	sys_statfs	N	Doesn't change system state
100	sys_fstatfs	N	Doesn't change system state
101	sys_ioperm	Y	Needed to trace process privilege level
102	sys_socketcall	Y	This is call multiplexes all socket calls. See next table 12 for details
103	sys_syslog	N	No need to duplicate this
104	sys_setitimer	N	Doesn't affect anything outside calling process
105	sys_getitimer	N	Doesn't change system state
106	sys_newstat	Y	Needed to trace file access
107	sys_newlstat	Y	Needed to trace file access
108	sys_newfstat	Y	Needed to trace file access
109	sys_uname	N	Doesn't change system state
110	sys_iopl	Y	Needed to trace process privilege level
111	sys_vhangup	Y	May be useful for analysis
112	defunct idle() call	N	
113	sys_vm86old	Y	Rare call that may be useful in analysis
114	sys_wait4	N	Does not provide information otherwise obtained
115	sys_swapoff	Y	Rare call that may be useful in analysis
116	sys_sysinfo	N	Doesn't change system state
117	sys_ipc	Y	Multiplexes all IPC calls. See table 13 for details
118	sys_fsync	N	Does not change state of system as seen by processes
119	sys_sigreturn	N	Doesn't affect anything outside calling process
120	sys_clone	Y	Needed to trace process tree

Table 8: List of system calls used in Linux, continued

#	Name	Log	Notes
121	sys_setdomainname	Y	Useful in analysis
122	sys_newuname	N	Doesn't change system state
123	sys_modify_ldt	N	Doesn't affect anything outside calling process
124	sys_adjtimex	Y	Rare event that could be useful in analysis
125	sys_mprotect	N	Doesn't affect anything outside calling process
126	sys_sigprocmask	N	Doesn't affect anything outside calling process
127	sys_create_module	Y	Important to record this, though any real implimenation of a logging system in the kernel would not actually allow dynamically insertable kernel modules
128	sys_init_module	Y	See above
129	sys_delete_module	Y	See above
130	sys_get_kernel_syms	N	Doesn't change system state
131	sys_quotactl	Y	Could be useful for analysis
132	sys_getpgid	N	Doesn't change system state
133	sys_fchdir	Y	Needed to trace file access
134	sys_bdflush	N	Shouldn't affect the state of the system as seen by processes
135	sys_sysfs	N	Doesn't change system state
136	sys_personality	Y	Could be useful after the fact
137	defunct afs_syscall	N	
138	sys_setfsuid16	Y	Needed to trace process privilege level
139	sys_setfsgid16	Y	Needed to trace process privilege level
140	sys_llseek	Y	Needed to trace file access
141	sys_getdents	Y	Needed to trace file access
142	sys_select	N	Does not provide information not obtained elsewhere
143	sys_flock	Y	Needed to trace file access
144	sys_msync	N	Only changes access method, not the status of the open file
145	sys_readv	Y	Recorded only for logRW runs. Needed to determine byte ranges accessed. The buffer read is not recorded.
146	sys_writev	Y	Recorded only for logRW runs. Needed to determine byte ranges accessed. The buffer written is not recorded.

Table 9: List of system calls used in Linux, continued

#	Name	Log	Notes
147	sys_getsid	N	Doesn't change system state
148	sys_fdatasync	N	Shouldn't affect the state of the system as seen by processes
149	sys_sysctl	Y	Useful for analysis
150	sys_mlock	N	Doesn't affect anything outside calling process
151	sys_munlock	N	Doesn't affect anything outside calling process
152	sys_mlockall	N	Doesn't affect anything outside calling process
153	sys_munlockall	N	Doesn't affect anything outside calling process
154	sys_sched_setparam	Y	May be useful in analysis
155	sys_sched_getparam	N	Doesn't change system state
156	sys_sched_setscheduler	Y	May be useful in analysis
157	sys_sched_getscheduler	N	Doesn't change system state
158	sys_sched_yield	N	Doesn't change system state
159	sys_sched_get_priority_max	N	Doesn't change system state
160	sys_sched_get_priority_min	N	Doesn't change system state
161	sys_sched_rr_get_interval	N	Doesn't change system state
162	sys_nanosleep	N	Doesn't affect anything outside calling process
163	sys_mremap	N	Doesn't affect anything outside calling process
164	sys_setresuid16	Y	Needed to trace process privilege level
165	sys_getresuid16	N	Doesn't change system state
166	sys_vm86	N	Doesn't affect anything outside calling process
167	sys_query_module	N	Doesn't change system state
168	sys_poll	N	Doesn't change system state
169	sys_nfsservctl	Y	Could be useful for analysis
170	sys_setresgid16	Y	Needed to trace process privilege level
171	sys_getresgid16	N	Doesn't change system state
172	sys_prctl	N	Doesn't affect anything outside calling process
173	sys_rt_sigreturn	N	Doesn't affect anything outside calling process
174	sys_rt_sigaction	N	Doesn't affect anything outside calling process

#	Name	Log	Notes
175	sys_rt_sigprocmask	N	Doesn't affect anything outside calling process
176	sys_rt_sigpending	N	Doesn't affect anything outside calling process
177	sys_rt_sigtimedwait	N	Doesn't affect anything outside calling process
178	sys_rt_sigqueueinfo	N	Doesn't affect anything outside calling process
179	sys_rt_sigsuspend	N	Doesn't affect anything outside calling process
180	sys_pread	Y	Recorded only for logRW runs. Needed to determine byte ranges accessed. The buffer read is not recorded.
181	sys_pwrite	Y	Recorded only for logRW runs. Needed to determine byte ranges accessed. The buffer written is not recorded.
182	sys_chown16	Y	Needed to trace file access
183	sys_getcwd	N	Doesn't change system state
184	sys_capget	N	Doesn't change system state
185	sys_capset	Y	Needed to trace process privilege level
186	sys_sigaltstack	N	Doesn't affect anything outside calling process
187	sys_sendfile	Y	Recorded only for logRW runs. Needed to determine byte ranges accessed.
188	defunct streams1() call	N	
189	defunct streams2() call	N	
190	sys_vfork	Y	Needed to trace process tree
191	sys_getrlimit	N	Doesn't change system state
192	sys_mmap2	N	Does not affect access to the file, just how it is done
193	sys_truncate64	Y	Needed to trace file access
194	sys_ftruncate64	Y	Needed to trace file access
195	sys_stat64	Y	Needed to trace file access
196	sys_lstat64	Y	Needed to trace file access
197	sys_fstat64	Y	Needed to trace file access
198	sys_lchown	Y	Needed to trace file access
199	sys_getuid	N	Doesn't change system state
200	sys_getgid	N	Doesn't change system state
201	sys_geteuid	N	Doesn't change system state
202	sys_getegid	N	Doesn't change system state

Table 10: List of system calls used in Linux, continued

Table 11: List of system calls used in Linux, continued

#	Name	Log	Notes
203	sys_setreuid	Y	Needed to trace process privilege level
204	sys_setregid	Y	Needed to trace process privilege level
205	sys_getgroups	N	Doesn't change system state
206	sys_setgroups	Y	Needed to trace process privilege level
207	sys_fchown	Y	Needed to trace file access
208	sys_setresuid	Y	Needed to trace process privilege level
209	sys_getresuid	N	Doesn't change system state
210	sys_setresgid	Y	Needed to trace process privilege level
211	sys_getresgid	N	Doesn't change system state
212	sys_chown	Y	Needed to trace file access
213	sys_setuid	Y	Needed to trace process privilege level
214	sys_setgid	Y	Needed to trace process privilege level
215	sys_setfsuid	Y	Needed to trace process privilege level
216	sys_setfsgid	Y	Needed to trace process privilege level
217	sys_pivot_root	Y	Rare event that is useful in analysis
218	sys_mincore	N	Doesn't affect anything outside calling process
219	sys_madvise	N	Doesn't affect anything outside calling process
220	sys_getdents64	Y	Needed to trace file access
221	sys_fcntl64	Y	Needed to trace file access
222	sys_tux	N	Only for use with TUX webserver

Table 12: List of system calls multiplexed through sys_socketcall

#	Name	Log	Notes
1	sys_socket	Y	Needed to trace socket connections
2	sys_bind	Y	Needed to trace socket connections
3	sys_connect	Y	Needed to trace socket connections
4	sys_listen	N	Does not provide information otherwise obtained
5	sys_accept	Y	Needed to trace socket connections
6	sys_getsockname	N	Does not change system state
7	sys_getpeername	N	Does not change system state
8	sys_socketpair	Y	Needed to trace socket connections
9	sys_send	N	Only passes information over connected sockets
10	sys_sendto	Y	Needed to trace socket connections
11	sys_recv	N	Only passes information over connected sockets
12	sys_recvfrom	Y	Needed to trace socket connections
13	sys_shutdown	Y	Needed to trace socket connections
14	sys_setsockopt	Y	Some options are needed to trace socket connections
15	sys_getsockopt	N	Does not change system state
16	sys_sendmsg	Y	Needed to trace socket connections
17	sys_recvmsg	Y	Needed to trace socket connections

Table 13: List of system calls multiplexed through sys_ipc

#	Name	Log	Notes
1	sys_semop	N	Only passes information along an established semaphore
2	sys_semget	Y	Needed to trace semaphore operations
3	sys_semctl	Y	Needed to trace semaphore operations
11	sys_msgsnd	N	Only passes information along an established queue
12	sys_msgrcv	N	Only passes information along an established queue
13	sys_msgget	Y	Needed to trace message queue operators
14	sys_msgctl	Y	Needed to trace semaphore operations
21	sys_shmat	Y	Needed to trace shared memory operations
22	sys_shmdt	Y	Needed to trace shared memory operations
23	sys_shmget	Y	Needed to trace shared memory operations
24	sys_shmctl	Y	Needed to trace shared memory operations

Table 14: List of non-system call events logged

Name	Description	Notes
lookup_start_log	Marks entry to path_walk	The look up flags are recorded at this point. The only relevant flag is LOOKUP_PARENT which tells us that the final file is not among those examined during the lookup.
lookup_log	Logs examination of an inode	This is logged primarily during a call to path_walk, recording each inode read to resolve a name. It is also recorded when a new file is created in any systemfile call. Only the inode number and device number are recorded for each inode.
lookup_return_log	Marks return from path_walk	The return value is recorded to note error. It also separates lookups during path_walk from the lookup logged for the creation of a new file.