A Linux Kernel Auditing Tool for Host-Based Intrusion Detection

William A. Maniatty, Adnan Baykal, Vikas Aggarwal,
Joshua Brooks, Aleksandr Krymer and Samuel Maura

Department of Computer Science
University at Albany
Albany, NY 12222

***Abstract*— Intrusion detection and response are critical in modern computing environments. In this paper, we focus on host-based intrusion detection, since once an intruder gains access it is very important to know what they did, how they did it and to formulate what recovery steps can be taken. The Linux kernel has historically had limited tools for generating and analyzing audit data. Since many intrusions are due to errors in design, implementation or configuration, we extend debugging based approaches to support intrusion detection. Our work in progress includes: (i) extensions to the Linux Trace Toolkit (LTT) which support a security audit, (ii) analysis tools for post processing this audit and (iii) using a flight recorder for back tracking responses to alarms. Our approach supports capturing the inputs, results and security credentials of each process at each system call. By focusing on the interface between user space and kernel space, we can characterize intrusions. We expect tight coupling of data mining with the kernel instrumentation to provide rapid feedback that enhances security and allows for refinement of both the instrumentation and mining approaches.**

I. Introduction

Intrusion detection and prevention are information intense activities which are increasingly important to systems administrators, modern police and investigative agencies. Security is maintained by monitoring audit logs. The quantity of traffic makes it mandatory for administrators and security experts to use intrusion detection systems (IDS), as tools for monitoring, summarizing and analyzing activity.

Intrusion detection systems are primarily of two types [6]: *signature-based*, in which audit data is searched for patterns known to be intrusive, or *anomaly-based*, in which aberrations from normal usage patterns are searched for. The former are susceptible to new attacks and hence reliance on these schemes is decreasing. While anomaly-based methods can detect new attacks, they are prone to higher false positive rates, as user behavior is often erratic and hard to model. Some methods [1] use a hybrid of the two to combine their benefits and counter their disadvantages.

A. Our Approach and Contributions

Our approach notes the similarity between debugging and intrusion detection and focuses on a novel approaches to extend a kernel debugging and performance analysis tool by combining audit trail generation and analysis. We developed kernel instrumentation as described in Section II which we want to integrate with data analysis to promote early detection and rapid response.

II. Augmented Linux Trace Toolkit

One of our goals is to maximize our software's impact, and we hope to be included in the next developer version (e.g. 2.7) of the kernel, for inclusion into the next production version, Linux 2.8 or 3.0 distribution.

LTT has dprobes [8] and kernel hooks [7], which support the insertion of probes in the kernel, by registering the invocation of exits (user defined callback functions) at any executable instruction.

Relayfs [9] supports rapid transfers of data between the kernel and the user space. Relayfs has an associated flight recorder option that uses a fast circular buffer, which can be dumped at opportune times (e.g. when a potential security violation occurs, we support dumping a detailed trace of system activity via flight recorder as seen in Section II-B).

This release of LTT is designed for kernel debugging and performance analysis, but prior to our instrumentation, did not gather sufficient information to generate a high quality security audit trail.

A. Overview of LTT Internals and Extensions

Historically, many intrusions occur because of programming errors permitting attacks such as buffer overflow [3], integer overflow [2], and race conditions. While it is desirable to find such errors during the design process or at compile time, systems and application errors are often only discovered at run time after code is released. Furthermore, configuration errors and insider threats are important classes of vulnerabilities that are hard, if not impossible to design around. Thus, intrusion detection and debugging both rely on having detailed knowledge of the system state and the ability to determine both the sequence of events and inputs that got the system into an undesirable state. In this section we describe our approach that extends a collection of tools to allow tracing kernel activity, both when the kernel is run directly on the hardware (native mode), and when the kernel is run as a guest operating system under a virtual machine (guest mode).

The Linux Trace Toolkit (LTT) instruments the kernel at compile or run time, and copying parameters and/or statistical

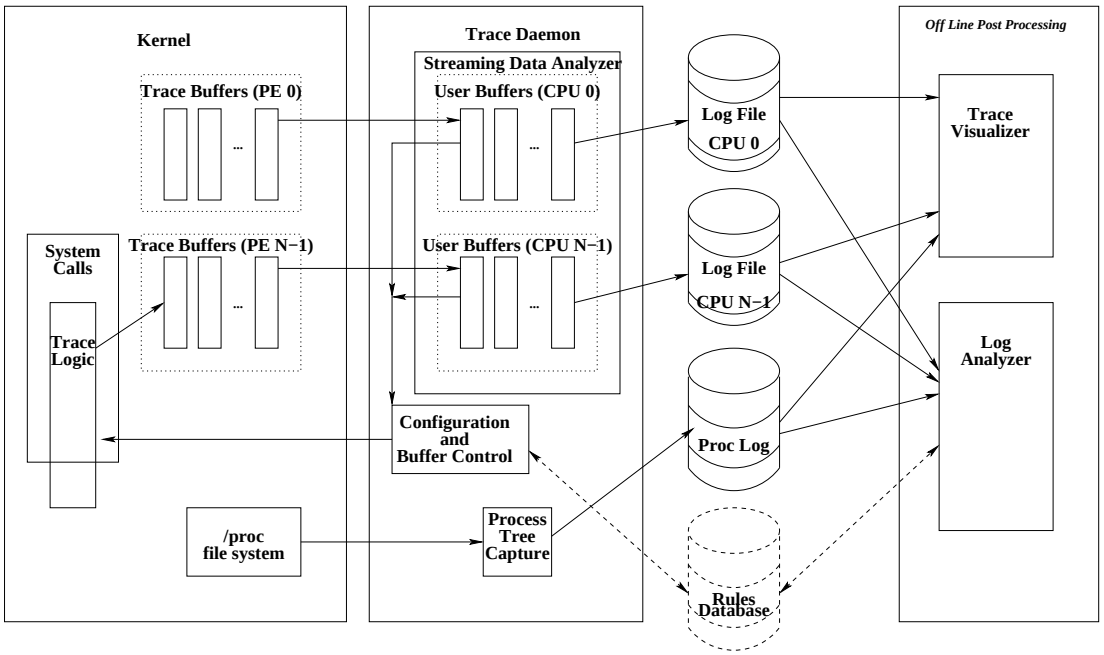

Fig. 1. Structure of our Augmented LTT Framework, Note the Rules Database is currently a work in progress

information into kernel buffers and interacting with a daemon process as shown in Figure 1. After making a system call to initiate data acquisition in the kernel, the LTT trace daemon records the process tree information from the /proc file system in the *Proc Log* file. In LTT data acquisition buffers are allocated on a per processor basis, with separate log files of data being acquired for each processor. Currently we use prereleases of LTT at the 0.9.6pre2 revision, much of our work is on the Linux 2.6.2 kernel.

Currently our work has focused on extending the logging instrumentation for a higher quality audit trail of system call events in LTT. When the buffers within the kernel fill with trace information, the kernel communicates with the daemon process by writing to a RelayFS channel which the daemon reads. The user space process then writes the buffers to disk, in log files. As these log files grow large very quickly with the additional data that we acquire, we use Gailly and Adler's zlib compression library [4], [5] for output, which frequently conserves on the order of 90% or more space.

Since we are auditing, we want to trace all interfaces between user processes and the kernel, thus using the Linux security module (lsm) hooks is not sufficient since:

1) We want to trace all system calls, even those that don't pass the initial parameter "sanity checks".
2) Many non-security oriented system calls, such as mremap and sbrk are vulnerable but lack lsm callback hooks.
3) Static compilation is more tamper resistant than loadable kernel modules.

B. Flight Recorder Back Trace

We used the flight recorder functionality of LTT in order to create a real-time privilege escalation detector. The credentials of each process, upon creation by the fork system call and each thread, created by the clone system call are stored into an array, privilege_table , indexed by the process id (PID). Later those values are compared to the credentials of the corresponding running process. Should they fail to match, our software generates a flight recorder dump. Due to the high rate of data acquisition, it may not be practical to log all system call activity in the default logs, but our approach records all events in the flight recorder buffers, which is relatively inexpensive as they are dumped infrequently. The idea is to respond immediately and catch the set of critical events that may have led to an intrusion. For each process created inside the kernel, a process_credentials data structure is initialized. That structure is then copied into a table indexed by the PID. A credential check occurs just before an event generated by a process is logged via ltt_log_event (the logging function). The private copy of the uid fields saved in the credentials table at process creation are compared to those of the currently executing process. The check can also be event specific, in other words whenever a process makes a particular system call closer attention can be paid to its credentials. We are currently verifying credentials for memory, filesystem and permission related system calls. However since we are logging all system calls, adding a credential check for every single event is not difficult.

C. Experiment Design and Apparatus

Our experiment was run on a dual processor AMD 1800+MP machine with 512 MB of RAM and a 250 GB disk with a UDMA 100 channel, running Fedora Core 2 Linux, with a modified Linux 2.6.2 kernel. For two consecutive weeks, we enabled tracing and logged normal user activities. In this

period of time, we gathered about 40GB of gzip compressed data. Our logs had about 28 sessions for our three most active users and fewer sessions for other users.

D. Efficiency of Instrumentation

In practice we observed between a 1% to 5% performance variation under the IOzone benchmark. We chose I/Ozone, since it generates a large number of system calls and would stress the I/O system during the time we were writing to the log buffers (in fact we used the same hard drive for logging and file access). Running the instrumented kernel with logging enabled caused a maximum slowdown of 5%, while not having logging enabled generally yielded a $1\% - 3\%$ performance boost (although we occasionally got a small slow down, probably due to some minor variation inherent in the benchmark's run time). Interactive and remote user testing indicates that there is no discernable difference in responsiveness, and observed performance appears comparable to running an uninstrumented kernel.

III. Conclusion and Future Directions

We have developed enhancements to the Linux kernel leveraging the features of open source tools, in particular, LTT, RelayFS, flight recorder. We are considering approaches to integrate the flight recorder dumps as response to user defined policy events. In practice, we have observed that under high load, the trace daemon may fail to empty the buffers quickly enough. We are investigating using either existing real time scheduling approaches or adjusting the scheduler so that the volume of recently logged data for each process is tracked as well as the number of full trace buffers in the kernel. One approach we are considering is to have the processor scheduler track the available trace buffer space in the kernel and detect when it drops below some threshold. At that time, the scheduler could both boost the priority of the trace daemon and reduce the priority of non real-time tasks that generate large numbers of events.

By carefully examining what sorts of data are available in the kernel and integrating with analysis approaches, we get fields that are security critical but are typically overlooked by other data acquisition tools. Anonymizing and sanitizing logs appears to be an open and significant challenge. We are currently exploring classification techniques, including ensemble learning approaches. We are interested in investigating data mining methods to suggest rules for policy based systems.

References

[1] Dorothy E. Denning: An Intrusion-Detection Model. IEEE Trans. Software Eng. 13(2): 222-232 (1987). An intrusion-detection model. *IEEE Trans. Software Eng.*, 13(2):222–232, February 1987.

[2] Wichert Akkerman. Userland can access linux kernel memory. BugTraq, published as a [Full-Disclosure] [SECURITY] [DSA-403-1] online at http://seclists.org/lists/bugtraq/2003/Dec/0010.html.

[3] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of DARPA Information Survivability Conference and Expo (DISCEX)*, 2000.

[4] P. Deutsch and J-L. Gailly. *ZLIB Compressed Data Format Specification version 3.3*. IETF, May 1996.

[5] J-L. Gailly, M. Adler, and G. Roelofs. On Line at http://www.gzip.org/zlib/.

[6] Sandeep Kumar and Eugene H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.

[7] R. J. Moore. Generalised kernel hooks interface - a high speed call-back mechanism for the linux kernel. In *Proceedings of the UKUUG Linux 2001 Linux Developers' Conference*, Manchester, England, July 2001. On line at *http://www-124.ibm.com/linux/projects/kernelhooks/*.

[8] R. J. Moore. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, USA, June 2001.

[9] K. Yaghmour. RelayFS: An efficient unified approach for trasmitting data from kernel to user space. In *Proceedings of the Ottawa Linux Symposium 2003*, Ottawa, Ontario, Canada, July 2003. The Linux Symposium's 2003 web page is at *http://www.linuxsymposium.org/2003/*.