Checkpointing and Logging for Intrusion Analysis and Recovery

W. Zhao, L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
wenbing@ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

1 Introduction

Although there exist many different strategies and techniques to prevent networked computer systems from being compromised, intrusions do happen due to operating system and application vulnerabilities.

Unfortunately, audit records produced through traditional system logging do not contain enough information to provide certainty as to how the system was compromised and what damage has been done. Such logs are even less useful to recover from an intrusion.

Recently, researchers have started to employ techniques that are used in the fault tolerance community, such as logging of external inputs and non-deterministic events, to replay the execution of the software, to facilitate better intrusion analysis [2].

2 The Need for Replay

Replay can be used for post-mortem forensic analysis of an intrusion, with the intrusion being monitored by logging and debugging tools and by a human expert. Analysis of the replay can disclose actions that the intruder deliberately deleted from, or modified in, the standard logs.

Replay can also be used to determine what was done by the intruder prior to the detection of the intrusion, even if the intruder has taken steps to conceal his actions, and can be used to undo those actions.

Replay can be used to perform a more careful and refined analysis to determine whether the potential intrusion is an actual intrusion or is an unusual but legitimate activity.

All intrusion detection is based on compromises between the diligence of the analysis and the overhead of the analysis, and on compromises between the rates of false negatives, which compromise system security, and false positives, which irritate the users. Replay allows an analysis with a low false negative rate and without irritating the users. Each detected potential intrusion can be replayed with a more detailed and discriminating analysis, or even with human supervision, to determine whether it is indeed a real intrusion or merely a false positive.

This research has been supported by the Multidisciplinary University Research Initiative in conjunction with the Air Force Office of Scientific Research under Contract F49620-00-1-0330.

3 Our Approach

In our approach, intrusion analysis is facilitated by deterministic logging and replay, and recovery from intrusion is carried out by rolling back to the last checkpoint before the intrusion occurred. Unlike [2], we use a direct-on-host, rather than a virtual machine-based, logging and checkpointing approach for several reasons.

Firstly, we have concerns about the performance overhead of the virtual machine. A virtual machine can easily slow down the execution of the applications by an order of magnitude or more. Moreover, many fundamental mechanisms used for deterministic logging [2] are feasible only if the virtual machine runs as a single-threaded process on the host operating system, which further reduces the application runtime performance.

Secondly, we believe that a system does not necessarily become more secure by using a virtual machine, because the weakness of a system is determined by its most vulnarable point. If the host operating system that supports the virtual machine is compromised, all the efforts and benefits of running a virtual machine are invalidated.

4 Deterministic Logging/Replay

To replay the execution of an application exactly as it has happened, external inputs, and their relative ordering, are recorded. With respect to a running application, external input into the application usually takes the form of library calls or system calls. Between two non-deterministic events, the application process runs "piecewise deterministically."

Because error returns can result from library or system calls, all calls of each thread are identified uniquely by a sequence number that is incremented for each library/system call within each thread. If the call is deterministic and it is successful, there is nothing to be done. If the call is non-deterministic and it is successful, appropriate information is recorded in a log. If the call is not successful, the error return, the error number, and the call sequence number are recorded in the log.

A process that contains n ($n \geq 1$) threads records events in $n+1$ logs. Each thread records events local to the thread in its own log. Events related to the operations on shared data are recorded in a centralized log. The log files are named appropriately with respect to the checkpoints taken during the execution.

On replay, the process is restored from the checkpoint of interest, and appropriate data structures are constructed and populated, based on the logs, before execution begins.

4.1 Multithreading

A prerequisite for deterministic replay is that accesses to shared data must be protected by mutexes (or semaphores). Before a thread is allowed to enter a critical section, it must claim the corresponding mutex.

The order in which threads issue mutex claim calls might lead to different state transitions. Therefore, the order in which different threads enter a critical section protected by a mutex are recorded for that mutex in the centralized log. Because different threads might enter different critical sections concurrently, a global mutex is used to protect the centralized log.

On replay, when a thread issues a mutex claim call, the centralized log is examined and the call is blocked if the mutex must be granted to another thread first.

4.2 Input/Output

The contents of the input buffer are recorded for replay. For output, even though it is not necessary to record the contents of the buffer, it is necessary to capture the response from the operating system. For example, the number of bytes written for the `write()` system call are recorded because it is not guaranteed that the full buffer will be completely written.

4.3 Physical Runtime Environment

Calls that access the physical runtime environment, such as the local clock (e.g., `gettimeofday()`) and process identifier (e.g., `getpid()`), are captured, and the information returned from the operating system are recorded. During replay, the logged clock value and process id are returned to the program, rather than the actual time and process id.

5 Checkpointing Applications

The state of applications of interest is checkpointed periodically. If the application involves multiple processes, the checkpoint of these processes must be coordinated. User-level or kernel-level process checkpointing [1] can be used.

In addition to the runtime image of the application state, files involved with the application are checkpointed using a copy-on-write mechanism [3]. Before a file is modified or deleted, a copy of the file is saved and tagged with a label consistent with a preceding application process checkpoint. Consequently, any file that is deleted by an intruder can be easily restored.

If a modification or deletion operation issued by an application on the checkpoint files is detected, such an operation is refused. Furthermore, the application is halted and appropriate actions are taken to inform the system administrator.

6 Conclusion

The deficiency of traditional logging mechanisms can be addressed by logging external inputs and non-deterministic events, so that execution of the system can be replayed exactly as it has happened, which can then be used to facilitate intrusion analysis using detailed and accurate information. The logging can be improved by periodically checkpointing the state of important processes and related files. With the checkpoints and logs, the system can be repaired by rolling back to a clean state before the intrusion occurred.

References

[1] W. R. Dieter and J. E. Lumpp, "User-level checkpointing for Linux threads programs," *Proceedings of the Freenix Track 2001 Usenix Annual Technical Conference*, Berkeley, CA (June 2001), pp. 81-92.

[2] W. Dunlap, S. T. King, S. Cinar, M. Basrai and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, Berkeley, CA (Dec. 2002), pp. 211-224.

[3] Y. M. Wang, Y. Huang, W. K. Fuchs, C. Kintala and G. Suri, "Progressive retry for software failure recovery in message-passing applications," *IEEE Transactions on Computers*, vol. 46, no. 10 (Oct. 1997), pp. 1137-1141