Correlating Multi-Session Attacks via Replay

Fareha Shafique, Kenneth Po, Ashvin Goel
Electrical and Computer Engineering
University of Toronto

Abstract

Intrusion analysis is a manual and time-consuming operation today. It is especially challenging because attacks often span multiple sessions which makes it is hard to diagnose all the damage caused by an attack. One approach for determining dependencies between the sessions of an attack is system-call taint analysis, but this method can generate large numbers of false dependencies due to shared objects such as a password file. In this paper, we propose a novel solution to this problem that replays sessions with tainted and untainted inputs and reasons about multi-session dependencies by comparing the session's outputs in the two cases. We present some initial experiments that show that this approach is promising and may allow building powerful intrusion analysis and recovery systems.

1 Introduction

Computer virus and worm attacks are increasingly being used to infect systems that are then used for criminal activities such as spamming services, denial of service and extortion. While some attacks are easily detectable (e.g, cause system crashes), most are "low level" or stealthy. These attacks often span multiple sessions (e.g., an ftp session followed by a login session) and can be difficult to pinpoint and correct. In corporate environments, the diagnosis may require that support services take down a system or a high-profile web site to fix the problem, which can be time-consuming and error-prone. In the meantime, the company site loses customer confidence, loyalty and revenue.

Our long-term goal is to design a system that allows fast and accurate post-intrusion analysis and recovery. We have implemented a recovery system called Taser [11] that can revert the file-system operations that are performed in a session, such as a login or ftp session. This system works well when an attack occurs in a single session and has a small footprint on the system (e.g, stealth attacks). However, attacks often span multiple sessions and correlating these sessions is a challenging problem. One approach for determining multi-session dependencies is system-call based taint analysis [14, 11]. In this method, for example, when a process B reads a file written by a tainted process A, then process B is also marked tainted. The intuition is that process B in the second session is causally related to process A in the first session via a file dependency. Unfortunately, this method causes false dependencies due to heavily shared objects, such as a password file that may have been written by an attacker but is read by all login sessions.

In this paper, we propose a novel method for reasoning about multi-session attacks. Our approach consists of two parts: 1) replaying sessions (or applications) with tainted as well as untainted versions of inputs, such as files, and 2) comparing the outputs of the sessions in the two cases to verify valid dependencies. Consider the password example above. We replay the sessions that started after the password file was modified by the attacker with a pre-modified version of the password file. Assuming that the attacker's following sessions depend on the password file modifications, these sessions will behave "differently" during replay (e.g., the attacker is unable to login). Conversely, the legitimate sessions will likely not depend on the modifications and will behave "similarly" during the replay and the original runs.

We detect differences (or similarities) in session behavior between the replay and the original run by observing outputs. The basic assumption in this approach is that at the point of replay, the attack is no longer latent, i.e. its effects can be detected in the session's outputs. These outputs can be defined in multiple ways. They can be application-independent such as the sequence of writes, sends or even system calls and/or the arguments issued by an application, or they can be application-dependant, such as the result of a computation. To improve the accuracy of detecting differences in behavior, we replay sessions multiple times with the original tainted inputs for training and then test with the pre-tainted inputs. If this test determines that the session behavior is different, the session is considered tainted with respect to the tainted input (e.g., password file modifications). This behavioral model for correlating multiple sessions is a black-box method that ignores the structure, operations or dynamics of applications. Instead, it determines dependencies between sessions based on correlating inputs and outputs.

This paper outlines the design of a taint verification framework for automatically determining an attacker's sessions based on replaying applications. The framework raises several issues: how should the replay be implemented so that it allows running applications with unperturbed or perturbed inputs, what inputs and outputs need to be captured, how is non-determinism in application behavior and interactions between applications handled, and how should the outputs be compared. We discuss these issues in the context of our framework and then present some initial analysis results. While we borrow well-known techniques from various areas such as recovery [15, 1, 18] and replay [5, 21, 19, 7, 4], the novelty of our approach is in combining these techniques with a view towards achieving our goal of automating post-intrusion analysis and recovery.

The rest of the paper describes our approach in more detail. Section 2 provides an overview of the Taser intrusion recovery system. Section 3 describes the design of our taint verification framework. Section 4 provides initial results to validate our approach. Sections 5 and 6 present some related work, conclusions and topics for future work.

2 Overview of Taser

We have designed and implemented a prototype intrusion recovery system called Taser [11] that supports analysis and recovery from intrusions, such as those caused by worms, viruses and rootkits. The Taser system recovers file-system data after an intrusion by reverting the file-system modification operations affected by the intrusion while preserving the modifications made by legitimate processes. The goal of recovery is to make the target system "intrusion-free" and, at the same time, not lose current work or data due to the recovery operations. This approach has similarities with system software upgrade where a buggy upgrade can be selectively rolled back without affecting the rest of the system. More details about Taser are available in our previous work [11, 10, 12].

Taser determines the file-system operations that were affected by an intrusion by deriving taint dependencies between kernel objects based on information flow that occurs as a result of system calls. This analysis starts with an externally supplied set of tainted objects known as detection points. Then the analysis uses dependency rules to taint other processes, files and sockets based on system-call operations such as read, write, fork, exec, and directory operations. For example, when a process reads from a tainted file, it is marked tainted. While this tainting approach is efficient, it is coarse grained and can cause significant false dependencies especially when objects such as the password file are heavily shared by applications. To reduce this problem, Taser provides tainting policies that ignore some of the dependency rules, and it also allows using whitelists to avoid tainting certain activities. However, it may require significant human effort to verify whether the policies yield correct results which can make this solution unsatisfactory.

3 Taint Verification via Replay

We propose verifying dependencies between applications (and sessions) by replaying applications and then observing whether perturbing an input to the application during replay changes the output of the application. In this black-box model, the application is assumed to be independent of the input perturbation if the output is unchanged.

Taser creates taint dependencies between processes, files and sockets based on reads and writes. A file or socket becomes tainted when a tainted process *writes* to the file or socket, and a process becomes tainted when it *reads* from a tainted file or socket. We verify dependencies caused by file read operations only. We assume that writes create a valid taint dependency, i.e. once a process is tainted then all its operations are considered tainted. We also assume that reading from a socket creates a valid taint dependency because sockets, unlike files, are transient and typically not shared across processes. Our experience shows that these assumptions do not cause false dependencies [11].

The verification process replays a process that originally read a tainted file with a pre-tainted version of the file. If the output of the process during replay is different from the original output, then we detect a taint dependency between the file and the process. The replay and the taint detection steps are discussed below.

3.1 Application-Level Replay

Currently, we are implementing a system-call based application-level replay system for taint verification. Our system will capture program input and output at the system call boundary and then replay a process by providing inputs captured during the original run [21, 19, 7]. Compared with whole system replay [5], an important benefit of this approach is that it allows replaying non-deterministically with perturbed application inputs [18]. When a process is replayed, our system will only provide network, file-system and user-input data that was read by the process during the original run (this data is available from Taser). The process will be allowed to run non-deterministically for all other inputs (e.g., system calls that do not return I/O data), which will allow replay with perturbed file inputs. This replay approach can fail (e.g., crash the program), but such a failure would be reflected as changed output and hence a valid dependency.

3.2 Taint Detection

Conceptually, a dependency between a tainted input file and a process is verified by replaying the process with a pre-tainted version of the file and then by comparing the original and the replay outputs of the process. If these outputs are different, then the process is assumed to depend on the tainted modifications to the input file. Unfortunately, this behavioral verification can lead to false dependencies because our replay method is inherently non-deterministic. It allows all non-read operations as well as interacting processes to proceed undirected, and as a result, the outputs can differ either due to the perturbed input file or due to other non-deterministic behavior.

To minimize the effects of non-deterministic behavior, we replay the process with the original tainted file multiple times for training and then replay the process with the pre-tainted file for testing. If the output of the testing run is significantly different from the outputs of the training runs, we detect a taint dependency between the tainted file and the process. Next, we describe a simple comparator tool we have implemented for taint detection.

Comparator Tool

The comparator tool replays a process or a session one or more times with the original inputs in training runs and then replays with the modified inputs for testing. Currently, it defines the output of a run as the trace of system calls issued by the application (all processes and subprocesses in the session) as logged by `strace`. This definition of output is simple to implement because it does not require defining output operations precisely. However, it introduces noise due to non-output operations (e.g., system call arguments that are inputs to the application) during comparison, and hence our evaluation results are conservative.

We use the comparator tool to calculate two ratios c_r and d_r to determine whether the testing run is significantly different from the training runs. The tool first calculates, for each iteration of training or testing run, two metrics c and d that are the total number of *common* and *different* lines between the output logs of the current iteration and the *original* run. Each line consists of a system call and its arguments and is considered common if the system call, its arguments (as recorded by `strace`) and its return value are exactly the same (or different otherwise). The comparator calculates the metrics using the `diff` program. Then, the training runs are used to calculate the min, max, mean and the deviation of the c and d values. Based on these values and the c and d values obtained during the test run, the comparator tool calculates the two ratios c_r and d_r as shown below:

$$\begin{aligned} c_r &= \quad c_{test}/c_{min} + c_{dev}/c_{mean} \\ d_r &= \quad d_{test}/d_{max} - d_{dev}/d_{mean} \end{aligned}$$

Ignoring the deviation and the mean values, when $c_r < 1$, it indicates that the test run has less in common with the original run than any of the training runs. Similarly, when $d_r > 1$, it indicates that the test run has more differences with the original run than any of the training runs. When both cases are true, the comparator tool considers the dependency to be valid. The deviation and the mean values ensure that variations in the training runs are taken into account while testing for a dependency.

4 Initial Evaluation

In this section, we present some preliminary results that validate our verification approach. These results were obtained by using our comparator tool. We are currently implementing and incorporating the replay system into our Taser recovery system and hence the replay itself was performed manually.

To test the feasibility of our idea, we performed two types of experiments, unit tests and a multi-session attack. The unit tests show the sensitivity of the metrics, while the multi-session attack shows that attack and legitimate sessions can be distinguished under realistic conditions. In all experiments, the applications were replayed four times for training. All the experiments that we performed and the results obtained for these experiments are presented below.

4.1 Unit Tests

We perform two types of unit tests. The first type of tests use the Ark 1.0.1 and Tuxkit 1.0 rootkits to determine whether backdoor programs (i.e., malicious programs that attempt to remain hidden from casual inspection) can be detected. The training data for the user-level Linux rootkits was generated using uninfected application binaries and the test runs were carried out using the trojaned binaries from the two rootkits. The Ambient RootKit (Ark) installs trojaned binaries for syslogd, login, sshd, ls, du, ps, pstree, killall, top, and netstat. These modified binaries allow an attacker to hide his presence from a naive system administrator. Our verification tests whether a user running these binaries versus the default binaries performs the same operations. The resulting c_r ratios for the programs ranged from 0.001 to 0.92 and d_r values ranged from 1.25 to 13.95. In all cases, running the trojaned binaries would be detected as a verified dependency. Tuxkit

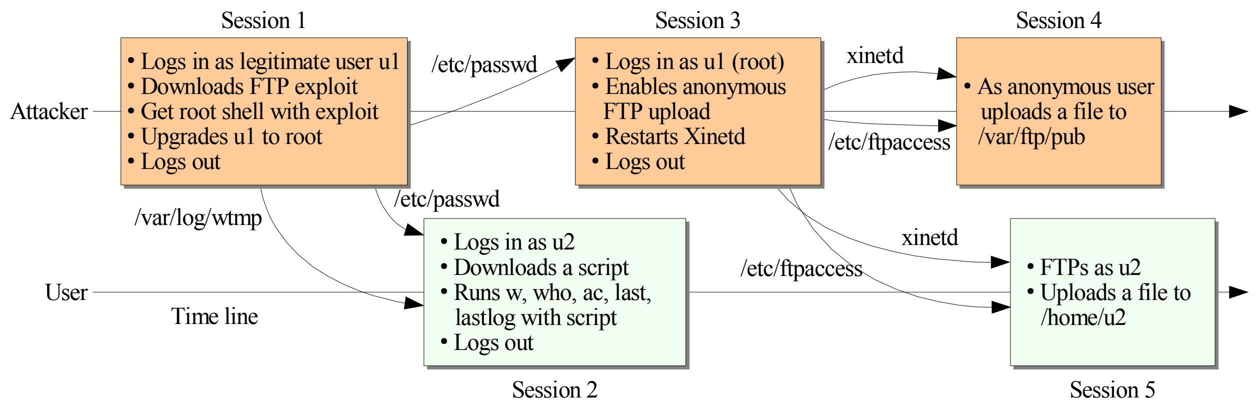

Figure 1: Multi-session attack scenario

also installs various trojaned binaries. We ran our experiments with all these binaries and the corresponding numbers were similar to Ark and ranged from $c_r = 0.01$ to 0.83 and $d_r = 1.58$ to 14.95.

The second type of tests vary inputs (e.g., user inputs, file inputs) to common services to determine whether our metrics are sensitive to these changes in input. We used the telnet/login, ftp, samba, and HTTP/Photo Gallery [16] network services for these tests. For the telnet/login daemon, we trained the program using a valid username and a password. Then we ran the program using an invalid username or an invalid password. In both cases, $c_r < 0.83$ and $d_r > 4.53$. For the ProFTPD FTP daemon, we trained the program using the same username, password and downloading of the same file as the original run. The test cases include downloading a different file (0.94, 8.11), using a different (VSFTPD) daemon (0.15, 4.2), minor version upgrade in Glibc (0.64, 2.88) and minor version change in ProFTPD (0.85, 2.3). We performed experiments using Samba also and obtained similar results. Finally, we conducted an experiment with a web server that hosts the Gallery photo album organizer [16]. The training consisted of several runs using the same username, password and photo file. The testing run uploads a different file (0.85, 5.16) and uses an updated Gallery version (0.49, 29.39). All these test cases would have established a dependency, which shows that our verification approach is sensitive to small variations in input.

4.2 Multi-Session Attack

The ftp server in this multi-session WU-FTPd attack is initially configured to run with anonymous reads enabled. The attack and a normal user's activity are shown in Figure 1. All logins in the figure (sessions 1, 2, and 3) are telnet sessions. For this attack, Taser by default would have marked all the sessions following the initial attacker's session as tainted because of dependencies caused by the `/var/log/wtmp`, `/etc/passwd`, `/etc/ftpaccess` files and the restarting of the `Xinetd` daemon. These dependencies are shown by arrows in the figure. During training, all sessions but the first session of the attacker (session 1) are replayed using the password file modified by the attacker. Then during the test run the password file is restored to the unmodified original and the same sessions are replayed. The results are shown in Table 1. These results show that the latter two attacker's sessions would have been marked as dependent on the attacker's first session while the two legitimate sessions would be considered independent (i.e., there are no false positives or negatives).

Session	**Result** (c_r, d_r)
Session 2, legitimate telnet session	(1.02, 0.93)
Session 3, attacker's telnet session	(0.7, 7.87)
Session 4, attacker's FTP session	(0.53, 1.99)
Session 5, legitimate FTP session	(1.02, 0.94)

Table 1: Results for multi-session attack scenario

4.3 Discussion

While the results for the tests we have performed are promising, our approach raises several concerns. Our replay method is non-deterministic because it allows varying inputs such as files, user-input and network I/O. However, timing and scheduling affects can introduce additional non-determinism that an attacker could use as "chaff" to vary outputs and confuse analysis. We plan to evaluate a larger class of applications to study these effects. In general, it should be possible to handle increased non-determinism with additional training.

A more serious concern relates to the robustness of our approach against an attacker that knows about our system. For example, an attacker could modify the system call log to counter our analysis. To reduce the risk of such intrusions, we use a kernel-level system-call logging system called Forensix and perform analysis on a separate, secured backend system [11, 10]. Forensix uses LIDS [23] to reduce the possibility of tampering with the kernel image or binary.

Finally, the choice of any deviation metric clearly affects which sessions are considered correlated. While these results may be inaccurate, they will still be useful for intrusion analysis where an investigator can quickly narrow in on pertinent activity and determine with further analysis whether any dependencies have been missed or are false. This final manual analysis can then be followed by accurate recovery.

5 Related Work

This work builds on a large body of literature related to recovery and replay, some of which is discussed below. Chen et al. [15, 2] explore the feasibility of failure transparent generic recovery and describe the relationships between the various recovery techniques that have been proposed. Rx [18] is a generic recovery technique that rolls back applications to a checkpoint and then, similar to this work, re-executes the program in a modified environment. Rx seeks to introduce non-determinism by inserting environmental changes such as padding buffers, zero-filling new buffers, or by increasing the length of a scheduling time-slot.

The Undoable E-mail Store [1] is an application-specific recovery method. It is based on three R's: Rewind, Repair, and Replay. In the rewind stage, the system is physically rolled back to a point before the failure. It is then repaired to prevent the problem from occurring again. The replay is logical and uses a proxy that requires application-specific knowledge to handle concurrency and visible outputs.

Replay has been used for various purposes including intrusion analysis and debugging. ReVirt [5] allows complete and precise replay at the machine level using a virtual machine monitor. This replay can be used to analyze intrusions and vulnerabilities in detail. King [14] uses ReVirt to implement analysis of intrusions via backtracking. A similar analysis method is used in our Taser system.

Flashback [21] provides fine-grained rollback and replay for debugging using system-call based replay. It uses shadow processes to efficiently rollback the in-memory state of a process and logs a process's interactions with the system to support deterministic replay. Liblog [7] is a user-level deterministic replay tool for debugging distributed C/C++ applications that does not require kernel patches and supports unmodified application executables.

The RolePlayer [4] is able to replay the client and the server sides of a session for a wide variety of application protocols when it is given examples of previous sessions. This proxy-based system captures and modifies identifiers such as the IP address and ports within packets during replay. These techniques should be helpful in replaying any network input to applications. Nagaraja et al. [17] provides an infrastructure for validating configuration changes prior to deployment. Their approach is also based on replaying the requests of network clients and comparing the server's responses as in RolePlayer.

While we use learning for intrusion analysis, there is a large body of related work on model-based intrusion detection [13, 8, 20, 6, 9]. These systems pre-compute a model of expected program behavior either dynamically or statically and use an execution monitor to verify that a stream of events, often system calls, generated by the executing process do not deviate from the model. The ability of the system to detect attacks with few or zero false alarms relies on the precision of the model. The initial approaches examined sequences of system calls without considering arguments. Sekar et al. [20] use a training phase to generate a program model that constrains system-call arguments and then use the model to safely execute untrusted code. Giffin et al. [9] construct static program models that are sensitive to the program environment.

Our current implementation uses a simple heuristic for detecting similarity between sessions. Cohen et al. [3] use signatures to detect and classify recurrent applicant bugs. PeerPressure [22] is a black-box approach that compares the Windows registry entries across a pool of Windows machines to identify configuration anomalies. Our comparator also takes a black-box approach. However we do not have access to machines running similar tasks, and hence we rely on replay to gather data for comparison.

6 Conclusions

This paper proposes using application-level replay and an input-output model of application behavior to determine related sessions of system activity and track the sessions of an attacker. We plan to use this information as input to the Taser intrusion recovery system to help accurately revert the effects of a multi-session attack. Our initial experiments show promising results, and as a result, we are currently implementing the replay system and incorporating it within Taser. At the same time, our approach raises several questions worth investigating: how long should applications be replayed to correlate sessions accurately, how does our non-deterministic replay approach affect accuracy, and how well does our system work with real intrusions.

Acknowledgments

We greatly appreciate the detailed feedback from the anonymous reviewers. We also wish to thank Alex Varshavsky, Zheng Li, Thomas Liu and several other members of the SSRG group in Toronto who provided comments on initial drafts of the paper.

References

[1] Aaron B. Brown and David A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the USENIX Technical Conference*, pages 1–14, June 2003.

[2] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 97–106, June 2000.

[3] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *Symposium on Operating Systems Principles*, pages 105–118, 2005.

[4] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the Network and Distributed System Security Symposium*, February 2006.

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

[6] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the USENIX Security Symposium*, pages 103–188, August 2004.

[7] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the USENIX Technical Conference*, pages 289–300, June 2006.

[8] Anup K. Ghosh, Aaron Schwartzbart, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *USENIX workshop on intrusion detection and network monitoring*, 1999.

[9] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-sensitive intrusion detection. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)*, pages 185–206, 2005.

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

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

[12] Ashvin Goel, Mike Shea, Sourabh Ahuja, Wu-chang Feng, Wu-chi Feng, David Maier, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. Poster Session in Proceedings of the Symposium on Operating Systems Principles, October 2003.

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

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

[15] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 289–304, October 2000.

[16] Bharat Mediratta. Gallery photo album organizer. `http://gallery.menalto.com/`, 2004.

[17] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 61–76, December 2004.

[18] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies — a safe method to survive software failures. In *Proceedings of the Symposium on Operating Systems Principles*, pages 235–248, October 2005.

[19] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the International Symposium on Automated Analysis-driven Debugging (AADEBUG)*, pages 69–76, September 2005.

[20] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the Symposium on Operating Systems Principles*, pages 15–28, 2003.

[21] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Technical Conference*, pages 29–44, June 2004.

[22] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 245–258, December 2004.

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