Trading Elephants for Ants: Efficient Post-attack Reconstitution

Meixing Le, Zhaohui Wang, Quan Jia, Angelos Stavrou,
Anup K. Ghosh, and Sushil Jajodia*

Center for Secure Information Systems,
George Mason University, Fairfax, VA
{mlep,zwange,qjia,astavrou,aghosh1,jajodia}@gmu.edu

Abstract. While security has become a first-class consideration in systems' design and operation, most of the commercial and research efforts have been focused on detection, prevention, and forensic analysis of attacks. Relatively little work has gone into efficient recovery of application and data after a compromise. Administrators and end-users are faced with the arduous task of cleansing the affected machines. Restoring the system using snapshot is disruptive and it can lead to data loss.

In this paper, we present a reconstitution framework that records inter-application communications; by logging only inter-application events, we trade our capability for data provenance and recovery *within* an application, for performance and the capability to recover long after the intrusion. To achieve this, we employ novel algorithms that compute the data provenance dependencies from the application interactions while minimizing the required state we maintain for system reconstitution. Our experiments show that our prototype requires two to three orders of magnitude less storage for recovery.

Keywords: Data Provenance, Causal Dependency, System Recovery.

1 Introduction

Computing has evolved into a necessary component for business, government, and military environments. Logistics, transportation, finance, intelligence, modern combat systems all depend on the correct operation of computer systems. Despite intense efforts towards improving software and network security, computers continue to be routinely compromised and exploited. Moreover, even when intrusions are detected, recovery happens long after the actual attack takes place.

* This work is sponsored in part by US National Science Foundation (NSF) grant CNS-TC 0915291 and AFOSR MURI grant 107151AA "MURI: Autonomic Recovery of Enterprise-wide Systems After Attack or Failure with Forward Correction." Sushil Jajodia and Meixing Le were partially supported by the National Science Foundation under grants CCF-103987 and CT-20013A, and by the Army Research Office DURIP award W911NF-09-01-0352. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

M. Rajarajan et al. (Eds.): SecureComm 2011, LNICST 96, pp. 460–469, 2012.

Administrators and end-users spend considerable time and effort "cleaning up" after the attacks. The standard practice consists of little more than re-formatting the disk, re-installing the operating system, and recovering user data from the most recent backup[1]. This is a time-consuming, error-prone process that is disruptive to end-users and enterprise operations.

The most recent work on system recovery is Retro [20]. Similar to Taser [5] and Solitude [7], Retro maintains an action history graph to capture the dependencies among system actor and objects (files) at multiple levels of abstraction. Contrary to all of these process-level recording approaches, our goal is to recover but also minimize the maintained system state by abstracting the low-level activities in the system trading-off recovery granularity. Instead of maintaining a voluminous log of low-level system activity including the input and return values of system calls, we attempt to simplify the causal dependencies which determines the log size. Our provenance graph maintains relations between objects only when those relations are necessary for potential future recovery. Last but not least, containers[2] abstract the execution of applications on physical machines.

Our approach attempts to lay a strong foundation to prevent cross-application contamination and provide efficient system reconstitution. To accomplish this, we designed a two-pronged architecture: on one hand, we record the data exchanged between contained applications; on the other hand, we leverage these logs to compute the application and user data provenance and use that information to recover. A challenging trade-off is the choice of the monitoring granularity. Using the finest possible granularity, the execution of every function call can be inspected and logged to obtain the most detailed knowledge of information flow. However, the computation and storage cost of recording and analyzing these events is prohibitively high for many applications and especially so over long periods of time. In contrast, many administrators rely solely on system and application logs for recovery. These logs usually offer a coarse-grain view that lacks sufficient information for analysis. Moreover, they are susceptible to tampering by attackers. To avoid these pitfalls, we only record application activities as container data exchanges. Therefore, we trade our capability for taint-tracking and recovery *within* a container for far lower processing and storage overhead.

As shown in experimental results, our system does not impose prohibitive logging or storage requirements: by selectively storing information based on data provenance, we provide better recovery for less storage when compared to pure versioning file systems, interval based backup or system snapshots. Finally, our recovery algorithm is able to reconstitute a typical desktop system even after the launch of hundreds of application instances long after the initial corruption. We demonstrate through different user studies, that the hourly temporary recovery log for a typical Desktop remains below 250MB and the persistent state is only

[1] For example, CERT's instruction on recovering compromised Unix and Windows NT systems: `http://www.cert.org/tech_tips/root_compromise.html`

[2] We use VEEs, VEs, and containers as abbreviations for Virtualized Execution Environments.

12MB for over 65 hours of collected data. This is between two to three orders or magnitude less information collected and stored compared to all prior research.

2 Related Work

The use of virtualization technologies for system monitoring and recovery has received a lot of attention [2,15]. In Revirt [2], a virtual machine snapshot encapsulates the entire system. By recording VM-to-host interactions the system stored a full OS-level replay of the entire duration of the attack. To enhance forensic analysis of intrusions, Goel *et al.* introduced Forensix [4], a system for forensic discovery and history reconstruction by monitoring a selected set of system calls. The Taser [5] recovery system was more geared towards tracking the propagation of an intrusion in a system, and it did not use virtualization to isolate processes. This allows intrusions quickly spreading in the entire system. Moreover, having to track all OS events, they generate enormous amounts of event data that have to be stored and analyzed. Solitude [7] used chroot jails that are limited mostly on the file system level. It cannot provide any strict kernel enforced isolation guarantees so that taint propagation through other channels such as memory, IPC is still possible. Contrary to Solitude, our approach enforces kernel-level separation so that isolate application instances bottom up in terms of memory, network, file system isolations.

Retro [20] re-executes the suspect actions to restore legitimate actions. It uses an action history graph to capture the dependencies among system actor and objects (files) at multiple levels of abstraction. Compared to Retro, we uses lightweight virtualization to encapsulate each application instance, and our aim is to minimize the maintained system state by abstracting the low-level activities in the system trading-off recovery granularity. We trade the recovery granularity for better performance compared to the 4-150GB per day for log storage. Apiary [12] used isolation on the file system and display layer to seamlessly isolate processes.

Taint analysis and system recovery using dependencies were also studied in [8,6,10,17]. There are other works [3,19] provide more accurate and efficient taint analysis, but all of them either incur high analysis overhead. Finally, researchers have used file versioning systems [13,11,18] to create file snapshots at block level to support recovery.

3 Threat Model and Isolation

3.1 Threat Model

Software vulnerabilities and the increasing installation of new applications and browser plug-ins are at the root of security risks. Malicious programs stealthily download and execute foreign code corrupting other files, sending out confidential information, etc. Most of these attacks are detected long after the initial intrusion take place. In our system, we use containers to isolate applications and

track their data communications over the entire duration of their life-cycle. We assume the attacks cannot break out of the container and corrupt the underlying system kernel as loading an kernel module in a container is prohibited. Furthermore, we assume that the point of intrusion (a tainted input) is provided to us by an external entity which can be an anti-virus or an intrusion detection system. We are also very conservative in marking tainted entities: containers become tainted after reading malicious files or receiving malicious network messages. All the output files and messages of a tainted container are considered tainted.

3.2 Container-Based Isolation

A container is a group of processes running on top of the same kernel as host within the same isolation zone. Starting up with a container template, an empty container will have all the necessary system processes of a working OS. These are all virtualized processes which is different from those on the host OS. This isolation is enforced by OpenVZ at kernel level. In our system, we put each application instance in a dedicated container. Process in one container can not communicate with or even be aware of the processes running in another container. The only allowed ways of data communication between containers is through networking or file sharing, and we record these events in our logs.

Inter Process Communication (IPC): For IPCs in our system, most of them such as *dbus* can be done within the isolated container. For X11 service IPCs, we choose to convert them to socket communications while all other inter-container IPCs are disabled. Therefore, for all IPCs in our system, either we do not trace them since they occur within one container, or we record them if they are the network events across containers. Without such isolation, it is easier for a process to taint another through IPCs, and it will be even worse if there is no mechanism to monitor these events on such a system.

Networking: Lightweight virtualization shares the same network processing code among containers but tags network related data (*i.e. packets, socket objects in kernel*) to achieve namespace isolation. Therefore, each VEE has its own independent network namespace. Namespace checks are enforced by kernel before any packet processing. Thus, network attacks that target applications or services running in one VEE won't affect the services running in the rest of the VEEs. With network isolation, each application instance has its own IP address.

Stackable File System: Unionfs [21] is a stackable file system service so that allow us to create one base template and share it among all containers, and this lowers the disk requirement. The base template is mounted to each container as read-only root "/" while a dedicated write-enabled layer is mounted on top of the root allowing each container to store its state in a separate directory. All the containers share one "shared_directory", therefore our system has the same functionality as the normal desktop systems. Whereas, all the interactions with this directory are monitored, and it is expected that containers only store persistent user data in this directory and all system related and temp files are stored within the containers' isolated file systems.

4 System Architecture

The overall system architecture is illustrated in Figure 1. Each application instance is running inside a VEE. We adopted the kernel probe [9] on the host OS to log system calls, and we monitor all the system call that can convey data between containers, which includes most file system related operations and network operations. The logs cannot be tampered by any process running in the containers.

4.1 Computing Provenance from Logs

The recorded system call logs offer a low-level view of all the container communications and data exchange including those with the host OS. Such view, however, does not immediately reveal the high-level and semantic dependencies among the containers. To produce the high-level view, we summarize and distill the raw system call entries into semantic objects (VEEs, files) and actions (read, write, overwrite, send via network). The summarized logs expose the logical events happening across the containers. We do the log summarization on the fly, and only keep the provenance information which will be discussed next as a high-level view of system events for recovery purpose. By doing this, we largely reduce the storage requirement for logs.

Here, we use the term **provenance** [16,1,14] to refer to the process of tracing and recording the origin of data and its propagation in the system. To be more precise, here we clarify what provenance means for file versions and the container states. The provenance of a file (at a certain version) is defined to be all the actions that modified any **present** portion of this file from its initial version. Modifications of sections that are not currently present (*i.e. discarded or overwritten*) are not part of the file provenance. On the other hand, the provenance of a container is the union of the provenance of all the input (files, network, and user input) of this container.

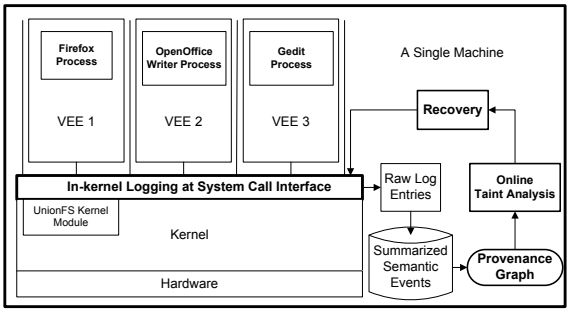

Fig. 1. Overall system architecture: application instances are confined inside containers. Logging, analysis, and recovery are performed on the same host.

Intuitively, the inputs of containers can be categorized into three types: reading shared files, receiving network messages, and user input. We assume that user input can be implicitly trusted since our protection is geared towards desktop users that have no incentive in harming themselves. We monitor the other two types of inputs.

4.2 Modeling States Using Provenance

By examining the provenance of files and containers, we can quantify a container's life-cycle into states. We use these states to track the containers' provenance set and create new file versions. To avoid unnecessary versioning, we further divide each container state into Input Sub-state and Output Sub-state. Each container state can only begin with new input, and output does not cause state transition. If the container receives continuous input events, it remains the Input Sub-state. The state of a container is changed only when the container is in the Output Sub-state and receives **new** input. An input is considered "old" if it has been already read in the past (for example, reading the same unmodified file twice). In case of old input, the provenance set does not need updating.

All containers are initiated from the same clean template. The container can become tainted only after it is potentially contaminated by some malicious or tainted input(s). Any tainted input causes a container to transit from the clean state to a tainted state. Since we do not know which input is malicious in advance, we treat all **new** inputs as the start point of a potential malicious input. For an input event, we check the provenance of this event given the state of the container. If all the previous information contributed to the event is already included in the container's provenance set, the container will not change its state since the provenance set will not change. If the input is a new event to the container, its state may be changed depending on whether the container is in Output Sub-state.

Traditionally, a new file version is generated whenever a container updates a file. Contrary, in our system, even if the container writes to the same file several times under the same state, we will only keep one version of this file. This is because, while we remain in the same container state, all file versions generated under this container state are either clean or tainted.

4.3 Recovery Using Provenance Graphs

Using the above model, the provenance of each file version is associated with a set of container states that have contributed to the content of that file. Therefore, files inherit the provenance of the container at that state. For the provenance of a container, subsequent states inherit the provenance set of previous states (in terms of time). Using these states, as well as the inherited relationships among them, we can construct the provenance graph of the system. In the provenance graph G=<V,E>, each node $v \in V$ represents a state of a container or a version of a file or a network message. Each edge $e \in E$ represents an input/output or state transition relation between the two nodes, which indicates a taint propagation path. Different states of the container are represented by the nodes in the graph, and they are connected by edges indicating the state transition. Each version of a file is a separate node also, and so are different messages. By traversing the graph in the opposite direction of the arrows, we can easily get the provenance of a file or message.

Because of the strong isolation provided by our system, the only possible ways of cross-container communication is through shared files and network communications. The provenance graph provides a concise representation of the container interactions enabling recovery even long time after the intrusion. The main idea is that, when given an initial intrusion point, we traverse the provenance graph to identify files and containers that have been tainted and require reconstitution from the latest recorded clean version.

5 Performance Evaluation

We implemented a fully working prototype of our system with OpenVZ. We performed several experiments in order to quantify the storage requirements of our system and the gains of using provenance for both storage and the capability to recover files when compared our system to both interval-based backup and pure file versioning. Our evaluation platform consisted of a 2.0GHz Pentium 4 CPU and 1GB of memory. The host OS was running CentOS 5 with a customized 2.6.24 kernel. OpenVZ containers are created from an Ubuntu 8.04 template.

5.1 User Study Using Real Deployment

We tested our prototype system under the load generated from typical desktop users. Five students were selected for the user study over a period of 7 days. The tested applications in VEEs include two web browsers (Firefox and Opera), two text editors (gedit and emacs), PDF reader (evince), and the Open Office suite (including writer, calc, impress, draw, math). In total, 218 VEEs were created in the experiments: 104 web browser VEEs, 47 Open Office VEEs, 40 text editor VEEs, and 27 PDF reader VEEs.

Our system transparently monitors the shared files without having to keep versions of files that are intermediate or non-persistent. For all the 87 shared files in our experiment (73 of which were downloaded from Internet), the total file size is about 52MB. Our system created 152 backup files for the 10 days operations, with a total size of 43MB. The size of raw system logs with all the system calls which is comparable to other systems is 13.1 GB. After preserving only the log entries about the shared directory and Internet activities, the log size is only 604MB. Finally, we only need 12MB of storage to maintain provenance information for recovery after 10 days operation. This is less than

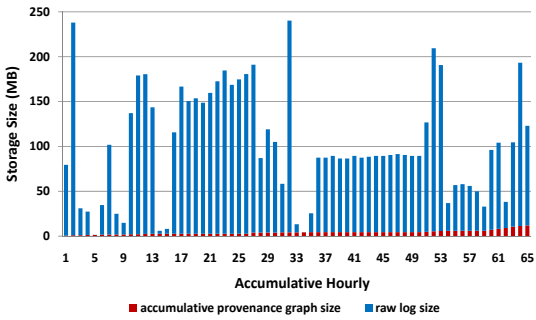

Fig. 2. Hourly Provenance Graph Generation

1/1000 of the original size. In this experiment, we generated the provenance graph from the non-summarized 604MB logs offline, it took 36 seconds to finish, which is mostly used to read the logs.

5.2 Hourly Provenance Graph Generation

In this experiment, we picked 65 hours of raw system logs in our user study. At the end of each hour, we ran our provenance generation algorithm, and updated the existing provenance graph with the new hourly information. Figure 2 depicts the storage space needed for hourly provenance graph generation. The maximum size of raw logs for one hour was less than 250MB. After each hour the analyzed raw logs were discarded, therefore, the total storage space for logs needed by our approach for 65 hours was still below 250MB. In addition, the bottom columns show the accumulative size of the provenance graph. As time pass, the size of the graph increased, however, even after 65 hours, the total size of the provenance graph was just around 10MB. From this figure, we can see the benefits of our approach in terms of state we have to maintain for recovery. For a typical system, it possible for us to recover data from an attack many days after the initial incident.

5.3 Versioning FS and Timed Backups

Here, we measured the storage overhead and the ability to recover information among interval-based backup systems, pure versioning file systems and our provenance-based approach. Interval-based backup approach takes periodic system snapshots, but the application and file information is lost from the last known good snapshot point. Pure versioning file systems keep every versions of files, so they require an enormous storage space. Our approach can always restore a corrupted file to the most recent clean version, if such exists. In contrast, interval-based backup can only partially recover files because it cannot differentiate between tainted and clean files after infection. Of course, the comparison of versioning and interval-based backup systems depends a lot on the system

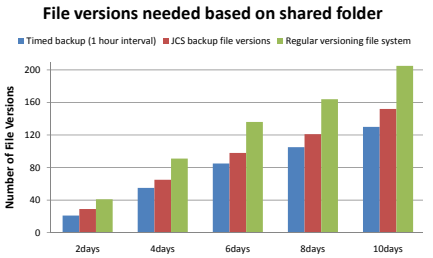

Fig. 3. Comparison of storage overhead for different backup approaches

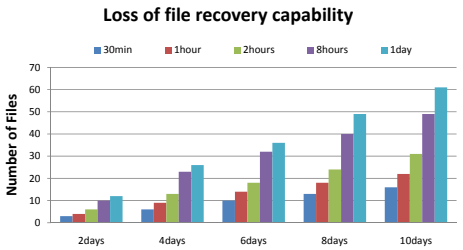

Fig. 4. Loss of file recovery capability for interval-based backup

usage but it is always the case that the versioning file system requires at least as much storage as the time-interval system. Both systems do not keep provenance information and thus cannot identify the proper versions of files to restore.

We compare the versioning storage overhead of shared files in terms of number of file versions. Figure 3 depicts the corresponding storage overhead comparison among interval-based backup, our approach, and regular versioning file systems. Using the provenance information, after 10 days, we can eliminate 53 versions of files compared to regular versioning file systems without losing any recovery information. The time interval based approach (1 hour interval) stored 22 versions less. Unfortunately, this difference in storage has an impact on the ability to recover files: Figure 4 shows the recovery ability lost in interval-based approach. We varied the backup time intervals to cover different backup scenarios. Although we were fairly aggressive in keeping data, for a 30 minutes interval, after 10 days, this approach lost 16 versions files, which means there are 16 possible cases that a tainted file can not be restored to the most recent clean version. Our results show that as we increase the time interval, less storage is required for backup. However, this diminishes the ability to recover data.

6 Conclusions

We presented a reconstitution framework that aims to provide fast and consistent recovery long after a corruption has taken place. We chose to log application events at the container level rather than the process-level offering a trade-off between finer-grain data recovery within an application for lower state requirements. We show through user studies, that the hourly temporary recovery log for a typical Desktop remains below 250MB and the persistent provenance graph is only 12MB for over 65 hours of collected data. To achieve this state reduction, we proposed a new method for generating data provenance graphs based on the state of the containers and interactions using files and network events. Recovery is feasible even after the launch of hundreds of desktop applications instances following the initial corruption.

References

1. Buneman, P., Khanna, S., Tan, W.-C.: Data Provenance: Some Basic Issues. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 87–93. Springer, Heidelberg (2000)
2. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: Revirt: Enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Systems Review (2002)
3. Goel, A., Farhadi, K., Po, K., Feng, W.-C.: Reconstructing system state for intrusion analysis. ACM SIGOPS Operating Systems Review (2008)
4. Goel, A., Feng, W.-C., Maier, D., Walpole, J.: Forensix: A robust, high-performance reconstruction system. In: 25th IEEE International Conference on Distributed Computing Systems Workshops (2005)

5. Goel, A., Po, K., Farhadi, K., Li, Z., de Lara, E.: The taser intrusion recovery system. In: SOSP 2005: Proceedings of the 20th ACM Symposium on Operating Systems Principles (2005)
6. Hsu, F., Chen, H., Ristenpart, T., Li, J., Su, Z.: Back to the future: A framework for automatic malware removal and system repair. In: ACSAC 2006: Proceedings of 22nd Annual Computer Security Applications Conference (2006)
7. Jain, S., Shafique, F., Djeric, V., Goel, A.: Application-level isolation and recovery with solitude. In: Eurosys 2008: Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems (2008)
8. King, S.T., Chen, P.M.: Backtracking intrusions. In: SOSP 2003: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (2003)
9. Krishnakumar, R.: Kernel korner: kprobes a kernel debugger. Linux Journal (2005)
10. Liu, P., Ammann, P., Jajodia, S.: Rewriting histories: Recovering from malicious transactions. Distributed Parallel Databases 8, 7–40 (2000)
11. Peterson, Z., Burns, R.: Ext3cow: a time-shifting file system for regulatory compliance. Transactions on Storage 1, 190–212 (2005)
12. Potter, S., Nieh, J.: Apiary: Easy-to-use desktop application fault containment on commodity operating systems. In: ATC 2010: USENIX 2010 Annual Technical Conference (2010)
13. Santry, D.S., Feeley, M.J., Hutchinson, N.C., Veitch, A.C., Carton, R.W., Ofir, J.: Deciding when to forget in the elephant file system. ACM SIGOPS Operating Systems Review 33, 110–123 (1999)
14. Seltzer, M., Muniswamy-Reddy, K.-K., Holland, D.A., Braun, U., Ledlie, J.: Provenance-aware storage systems. In: USENIX ATC 2006: Proceedings of the USENIX Annual Technical Conference (2006)
15. Sharif, M., Lee, W., Cui, W., Lanzi, A.: Secure in-vm monitoring using hardware virtualization. In: CCS 2009: Proceedings of the 16th ACM Conference on Computer and Communications Security (2009)
16. Simmhan, Y.L., Plale, B., Gannon, D.: A survey of data provenance techniques. Technical report, Computer Science Department, Indiana University, Bloomington IN 47405 (2005)
17. Sriranjani, S., Venkatesan, S.: Forensic analysis of file system intrusions using improved backtracking. In: IWIA 2005: Proceedings of the Third IEEE International Workshop on Information Assurance (2005)
18. Soules, C.A.N., Goodson, G.R., Strunk, J.D., Ganger, G.R.: Metadata efficiency in versioning file systems. In: FAST 2003: Proceedings of the 2nd USENIX Conference on File and Storage Technologies (2003)
19. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: ASPLOS 2004: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (2004)
20. Taesoo Kim, N.Z., Wang, X., Kaashoek, M.F.: Intrusion recovery using selective re-execution. In: OSDI 2010: Proceedings of the 9th Symposium on Operating Systems Design and Implementation (2010)
21. Unionfs, `http://www.am-utils.org/project-unionfs.html`