WatchIT: Who Watches Your IT Guy?

Noam Shalev[†] Idit Keidar[†] Yosef Moatti[*] Yaron Weinsberg[*]

Technion, Israel Institute of Technology[†]
IBM Research, Israel[*]

{noams@campus,idish@ee}.technion.ac.il {moatti,yaron}@il.ibm.com

ABSTRACT

System administrators have unlimited access to system resources. As the Snowden case shows, these permissions can be exploited to steal valuable personal, classified, or commercial data. In this work we propose a strategy that increases the organizational information security by constraining IT personnel's view of the system and monitoring their actions. To this end, we introduce the abstraction of *perforated containers* – while regular Linux containers are too restrictive to be used by system administrators, by "punching holes" in them, we strike a balance between information security and required administrative needs. Our system predicts which system resources should be accessible for handling each IT issue, creates a perforated container with the corresponding isolation, and deploys it in the corresponding machines as needed for fixing the problem.

Under this approach, the system administrator retains his superuser privileges, while he can only operate within the container limits. We further provide means for the administrator to bypass the isolation, and perform operations beyond her boundaries. However, such operations are monitored and logged for later analysis and anomaly detection.

We provide a proof-of-concept implementation of our strategy, along with a case study on the IT database of IBM Research in Israel.

1. INTRODUCTION

A system administrator position is typically characterized by unlimited permissions and access to system resources. The Edward Snowden [9] case has brought up again the discussion regarding the bulk permissions given to system administrators, which can be exploited to steal valuable, classified, or commercial data.

The current trend in cloud systems and development environments is to deploy containers, which are sets of processes that are isolated from the rest of the machine. Containers provide isolation that resembles the one provided by VMs, but with less overhead. Seemingly, the isolation concept that containers provide goes against the very nature of system administration – indeed, IT personnel get permissions for a reason.

In this work we propose to exploit container properties while, counter intuitively, perforating its isolation in order to create a middle-ground which is useful enough for administrators and yet controlled. As a result, our container-based strategy constrains IT personnel's view of the system, provides a way to monitor their actions, and yet preserves their superuser privileges. The solution we devise is based on two principles: (1) isolating the system administrator from presumably irrelevant resources, and (2) enabling her to perform actions beyond the isolation boundaries, under monitoring and logging. For this purpose, we introduce the notion of *perforated container* – a container that shares at least one of the host's available namespaces.

Our proposed strategy first approximates which system resources should be accessible for handling each type of *ticket* that the IT department deals with. Then, for each ticket category, we create a perforated container, whose permissions and isolation are determined according to the corresponding prediction. When handling a ticket, our system deploys the corresponding perforated container on the target machines. The IT person can then log into the perforated container and operate from within it while being confined to the namespace limitations dictated by the container, and at the same time having a superuser privileges. Though compartmentalized from system resources deemed irrelevant, when necessary, the system administrator can ask a dedicated software installed on the host (*permission broker*) to perform operations on the compartmentalized resources on his behalf. In such cases, these actions are monitored and logged, thus enabling future analysis and anomaly detection.

We implement a framework for using our approach. The framework includes a program that serves as the permission broker and a software system for getting a free text issue description, classifying it, and deploying a corresponding perforated container at the target machines. We further present a case study on the IT database of IBM Research in Israel: We divide actual IT tickets into a few main categories and classify them using common machine learning algorithms. Finally, we adjust a perforated container to each ticket class.

To conclude, our contributions in this paper are as follows:

1. We devise a novel container-based approach to protect from insider superuser threats.
2. We provide a proof of concept implementation of our approach.
3. We present a real world case study.

MIST'16, October 28 2016, Vienna, Austria

© 2016 ACM. ISBN 978-1-4503-4571-2/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2995959.2995968

2. BACKGROUND

2.1 Containers

Modern Linux-based containers utilize the Linux *namespaces* mechanism in order to provide isolation between the processes in the system. There are six types of namespaces, which provide per-process isolation of the following operating system aspects: file-system (MNT), UNIX time-sharing system (UTS), inter-process communication (IPC), process IDs (PID), network (NET) and user IDs (UID). A traditional Linux container associates all the processes that it contains with a new namespace of each available type, thus providing an isolated environment with the salient benefits of a VM.

That being said, it is possible to associate a process with only a subset of the available namespaces, thus isolating the process from only a subset of the system resources. For example, one can associate a process only with a new PID namespace, preventing it from seeing other processes in the system; however, since it shares the rest of the host's namespaces, it is able to see the host's file-system, use its network stack, etc. In this work, we use the term *perforated container* to refer to a group of processes that has reduced isolation, namely, a container that shares a namespace or part thereof with the host.

2.2 Typical IT Work Flow

IT work-flow usually consists of three main steps: (1) The end user fills a ticket, usually as free text, which describes the problem that needs to be fixed. (2) The IT department receives the ticket, reads it, and manually dispatches the ticket to the appropriate IT specialist. (3) The IT specialist gains access to the computer in question, usually remotely, and attends to the problem under superuser privileges on the target machine.

The last step constitutes a major security breach, as while attending the ticket, the IT specialist has unmonitored access to all of the employee's system resources, while most of these are irrelevant to the ticket handling. This can be exploited to steal classified commercial data, copy personal information, or install malware on the end-user's computer.

3. WATCHIT

In this section we present our proposed system, *WatchIT*. We begin in Section 3.1 with an overview of WatchIT structure. Next, in Section 3.2 we elaborate on the permission broker and its advantages. Finally, in Section 3.3 we describe our proposed IT framework. We assume that all machines in the system have some container software installed, such as Docker [8] or LXC [1].

3.1 Architecture Overview

The architecture of our proposed system is depicted in Figure 1. The work-flow begins with the submission of a request to the IT department by an end-user; we refer to this request as a *ticket*. Tickets are written in free text, and include complaints about bad configurations, connectivity problems, software bugs, expired licenses, insufficient permissions, etc.

The tickets are submitted to an organizational *IT framework* that manages ticket handling. The IT framework analyzes the tickets and classifies each of them to one of many

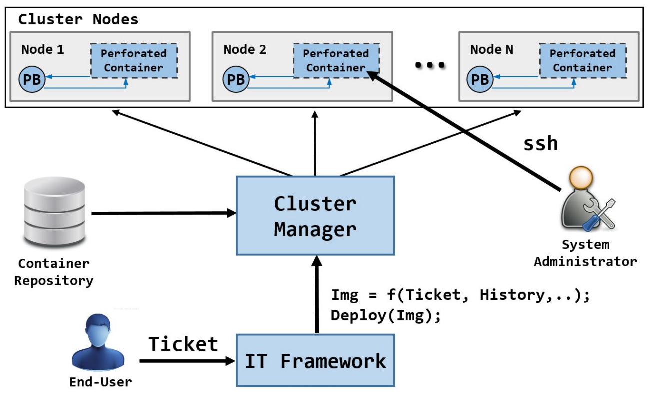

Figure 1: WatchIT architecture overview.

predefined classes. Each class is associated with a collection of problems, and a set of permissions for attending to them. Therefore, for each ticket class, we assign a different perforated container, each of which is associated with the corresponding set of configurations, such as namespace separations and installed software, and has sufficient permissions for handing the corresponding ticket. Similarly to the Docker architecture [8], the various container images are held in a dedicated image repository for quick deployment. Upon classifying the ticket, the framework asks the cluster manager (such as Apache Mesos) to deploy the corresponding image on the target machines.

Following the deployment, IT personnel can then log into the deployed containers in the target machines, and attend to the ticket. Thanks to the isolation provided by the namespace subsystem of the OS kernel, the administrator is confined to the limitations dictated by the perforated container, and cannot operate on system parts that the container is not exposed to.

3.2 Permission Broker

On some occasions, the permissions assigned to a container may be insufficient for handling the ticket. To address such cases, we augment each machine in the system with a *permission broker*, PB in Figure 1. The permission broker is a software component installed on the host file-system, which is able to grant a running container additional permissions, and provide it with information regarding the host system, while logging all accesses. For example, if the IT specialist attends to a network problem and during the work would like to see the list of running processes in the system, an information from which he is compartmentalized, he can submit a request to the permission broker to obtain the answer. The permission broker, in turn, can answer the request if it follows the security policy, and can refuse otherwise. In any case, these requests are logged, and can be monitored and analyzed later for anomaly detection. Moreover, this information can be used to further adjust the configurations of future perforated containers: if, for example, the same ticket class asks frequently to see the list of running processes in the system, then we might decide to either remove the PID namespace isolation from the corresponding perforated container or split the ticket class into two subclasses, where one of them has no PID isolation.

Note that if the permission broker is configured to answer

```
root@ee9b40a99dfa:/home/itsupport# ps -a
  PID TTY          TIME  CMD
    6 ?        00:00:00  bash
   71 ?        00:00:00  testscript
   72 ?        00:00:00  sleep
   73 ?        00:00:00  ps
root@ee9b40a99dfa:/home/itsupport# PB ps -a
  PID TTY          TIME  CMD
 1075 pts/28   00:00:00  sudo
 1077 pts/28   00:00:00  docker
 1139 pts/17   00:00:00  bash
 1194 pts/14   00:00:00  runServer
 1195 pts/14   00:00:00  java
 1272 pts/17   00:00:00  testscript
 1273 pts/17   00:00:00  sleep
 1276 pts/19   00:00:00  ps
root@ee9b40a99dfa:/home/itsupport#
```

Figure 2: Example of using the permission broker.

all requests, then the administrator effectively has unlimited permissions. However, all such unanticipated requests are logged and hence may be monitored.

As a consequence, the permission broker's log is amenable to anomaly detection. Anomaly detection has been widely researched within diverse application domains [6] while one of the biggest challenges in this field is handling enormous amounts of data.

Our proposed permission broker logs only IT activities that diverge from the predefined permissions, which hopefully capture the reasonable behavior for handling a specific ticket. Therefore, our log essentially employs a filter that may drastically reduce the amount of collected data compared to logging all administrator actions. Moreover, when enough data is collected, a better fit of perforated container to ticket is made possible, thus further reducing the amount of gathered data, and further easing future data analysis.

3.3 IT Framework

Our proposed IT framework performs three main steps for each ticket: (1) Log the ticket in a database for later processing. (2) Analyze the ticket content and classify it to one of the predefined categories. A classifier may be built using machine learning or simpler algorithms. (3) Request the cluster manager to deploy the most suitable container (among available ones) on the target machines. Note that the second step requires preprocessing, which should be done periodically. The preprocessing includes surveying the ticket categories and adjusting a container with sufficient permissions for each ticket class, considering security requirements and IT work-flow.

4. IMPLEMENTATION

For our proof-of-concept implementation, we build a computer cluster consisting of five nodes, each running Ubuntu 14.04. We designate one node to serve as the IT framework; one node to serve as a repository for the perforated containers; and three nodes as user machines. Each user machine is installed with our permission broker software, and has an `itsupport` user account. We use Docker as our container software, and Docker Swarm for deploying the containers at the target machines. We configure five perforated containers and build their corresponding images according to the case study presented in the next section.

We implement the permission broker in Java, using a client-server architecture, where the server side is installed on the host, and the client side is installed on the container. The communication goes through the tcp/ip stack; for serializing the data we use Google's Protocol Buffers; and all requests are logged in a MySQL database. Note that this design exposes the permission broker to other insider and outsider attacks, hence we use SSL for securing the container-host communication.

Figure 2 depicts a typical use case of the permission broker. The IT specialist is logged into a perforated container, and can freely operate within its namespace limitations. In the figure, we see that when executing "`ps -a`" from the container, the administrator sees only the processes that belong to the current PID namespace. However, if the IT specialist wishes to get information about all processes on the host (which is beyond the allowed scope), he must go through the permission broker. Calling the permission broker is done by prefixing the command with `PB`. This simple API asks the permission broker to execute the command on behalf of the compartmentalized administrator. Note that the permission broker does not have to respond to all execution requests. It works using a black/white list of commands, which is managed manually; in any case, all the requests that are sent to the permission broker are logged for later analysis.

5. CASE STUDY

5.1 Ticket Analysis and Grouping

For our case study, we analyze the IT database of IBM Research in Israel, containing about 33,000 tickets which were collected during the years 2009-2015. Tickets are written in free text by the end-users and are manually sent by the IT department to the corresponding IT specialist. From this corpus, we gather the tickets pertaining to Linux machines. The filtering is done by choosing only the tickets that were assigned to IT personnel that specialize in Linux issues; this leaves us with a corpus containing around 4000 tickets.

In order to cluster the tickets into categories, we use topic modeling [3]; such algorithms take a corpus and group the words across it into topics. Before performing the topic modeling, we pre-process the corpus by applying word stemming, stop word removal, and deletion of repeated words that do not add information (like 'hello' and 'please'). We then use Latent Dirichlet Allocation (LDA) [4] to process the data and group it to topics. We ask LDA to perform a coarse-grained grouping of the data to five topics, which is sufficient for our proof-of-concept.

Partial results of running LDA appear in Table 1. LDA represent a topic as a distribution over words; thus, the full results of running LDA outputs for each topic a list of all the words that appear in the corpus. Each word in each list is associated with a number that represents the likelihood for finding that word in a text on the corresponding topic. For illustration purposes, Table 1 shows for each topic only the top-five words that match it, along with the number that represents the probability that an instance of this word associates it with the corresponding topic.

By examining only the top five words of the results, one can infer the general picture: Topic A mainly refers to installing new software on a server/machine; topic B refers to problems with Matlab; topic C refers to connection and ssh problems; topic D refers to user/password problems; and topic E mainly refers to issues regarding process management and server maintenance.

Topic A		Topic B		Topic C		Topic D		Topic E	
install	0.046	matlab	0.045	connect	0.030	access	0.023	run	0.018
server	0.025	license	0.032	access	0.014	machine	0.017	server	0.016
linux	0.021	install	0.014	server	0.013	password	0.015	reboot	0.012
machine	0.016	user	0.014	ssh	0.012	server	0.015	job	0.009
version	0.014	toolbox	0.014	error	0.012	user	0.012	load	0.009

Table 1: Partial results of running Latent Dirichlet Allocation on our dataset.

5.2 Permission Assignment

Given the above, we build five perforated containers, which differ in their permissions and resource isolation. The list of permissions for each ticket class appear in Tables 3 and 2. For example, for attending class B tickets (Matlab problems) – the corresponding perforated container gets access to Matlab related folders (vi), and to the home folder of the user (vii), where additional configuration files may reside; however, this container is compartmentalized from other system resources, such as seeing other running processes on the system (PID namespace), the network, the rest of the file system, etc. On the other hand, for attending class E tickets (process management issues), where the administrator might need to add users, kill processes, or even reboot the machine, the corresponding perforated container shares the UID, PID, and IPC namespaces of the host; no specific access to the host's file-system is granted in such case.

Note that these per-container lists of permissions are subject to the jurisdiction of the IT supervisors. The content of these lists, as well as the number of ticket classes can easily be tuned for the needs of the organization. For issues that do not match any of the classes we create a fully isolated container, thus tracking and logging all the operations that are done while attending to the ticket.

6. RELATED WORK

Current solutions [2, 5] for providing protection from insider threats are mainly based on mandatory access control and mostly aimed against privileged accounts of rogue employees. However, they all trust the system administrators.

To the best of our knowledge, the only previous work which is intended for confining system administrators is the Jail [7] mechanism that appears in FreeBSD distributions. Jails provide the ability to create multiple root users, each with a different view of the system. This technique is intended for web service providers that would like to grant their customers root privileges, while protecting the files and services of one customer from access by any other customer. Contrary to our approach, in [7], the system view of each root user is determined beforehand and does not change with the needs, no actions can be performed on the host from within the Jail, and monitoring is not provided.

ID	Permission
(i)	Host MNT namespace
(ii)	Host NET namespace
(iii)	Host UID namespace
(iv)	Host PID namespace
(v)	Host IPC namespace
(vi)	Access Matlab related folders
(vii)	Access user home folder

Table 2: List of possible permissions

7. CONCLUSIONS

We propose an approach for mitigating attacks from powerful insiders - the organizational IT department. Our strategy exploits container's properties, thus preserving the superuser permissions while confining them at the same time. We further implemented a proof-of-concept of our approach and provide a case study on a real IT database.

We have presented in this paper what we believe is the best configuration. That being said, each organization that adopts our approach can tune it and adjust its settings according to its needs. For example, one can log all IT activities instead of just those that go through the permission broker; apply online anomaly detection; create more ticket classes etc.

8. ACKNOWLEDGMENTS

This research was funded in part by the Hasso-Plattner Institue (HPI) and the Technion Funds for Security Research. We thank the IBM Research IT department for granting us access to their database.

9. REFERENCES

[1] *Linux Containers*, 2016. Avaliable at https://linuxcontainers.org/.
[2] M. Bauer. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux J.*, 2006(148):13–, Aug. 2006.
[3] D. M. Blei. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, Apr. 2012.
[4] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.
[5] M. Bugliesi, S. Calzavara, R. Focardi, and M. Squarcina. Gran: Model Checking Grsecurity RBAC Policies. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 126–138, June 2012.
[6] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.
[7] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *In Proc. 2nd Intl. SANE Conference*, 2000.
[8] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), Mar. 2014.
[9] J. Verble. The NSA and Edward Snowden: Surveillance in the 21st Century. *SIGCAS Comput. Soc.*, 44(3):14–20, Oct. 2014.

Ticket Class	Permission Set
A – Installing new software	(i)
B – Matlab	(vi),(vii)
C – Communication	(ii),(vii)
D – Users/Passwords	(vii)
E – Process management	(iii),(iv),(v)

Table 3: Permission set for each class.