WatchIT: Who Watches Your IT Guy?

Noam Shalev*
Technion, Israel Institute of Technology
Haifa, Israel
noams@technion.ac.il

Idit Keidar
Technion, Israel Institute of Technology
Haifa, Israel
idish@ee.technion.ac.il

Yaron Weinsberg
Microsoft
Herzliya, Israel
yaronwe@microsoft.com

Yosef Moatti
IBM Research
Haifa, Israel
moatti@il.ibm.com

Elad Ben-Yehuda
IBM Research
Haifa, Israel
eladby@il.ibm.com

ABSTRACT

System administrators have unlimited access to system resources. As the Snowden case highlighted, these permissions can be exploited to steal valuable personal, classified, or commercial data. This problem is exacerbated when a third party administers the system. For example, a bank outsourcing its IT would not want to allow administrators access to the actual data. We propose WatchIT: a strategy that constrains IT personnel's view of the system and monitors 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. Following the principle of least privilege, 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 as needed for fixing the problem.

Under this approach, the system administrator retains superuser privileges, however only within the perforated container limits. We further provide means for the administrator to bypass the isolation, but such operations are monitored and logged for later analysis and anomaly detection.

We provide a proof-of-concept implementation of our strategy, which includes software for deploying perforated containers, monitoring mechanisms, and changes to the Linux kernel. Finally, we present a case study conducted on the IT database of IBM Research in Israel, showing that our approach is feasible.

CCS CONCEPTS

• **Security and privacy → Security services**; **Systems security**;

KEYWORDS

Perforated Container, Privileged Insider Threat

ACM Reference Format:
Noam Shalev*, Idit Keidar, Yaron Weinsberg, Yosef Moatti, and Elad Ben-Yehuda. 2017. WatchIT: Who Watches Your IT Guy?. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 16 pages.
https://doi.org/10.1145/3132747.3132752

1 INTRODUCTION

According to IBM's annual cyber-security report [21], 44% of cyber attacks detected last year were carried out by malicious insiders; this positions malicious insiders as one of today's biggest security threats. Among a company's employees, IT workers pose the greatest risk to the organizational information security, since a system administrator typically has unlimited permissions and access to system resources. The famous Edward Snowden [47] case, among others [17, 26], has again brought to public attention the danger arising from giving bulk permissions to system administrators, which can be abused by greedy or disgruntled employees to steal valuable, classified, or commercial data.

The problem is aggravated in cases where a third party handles the organizational IT. For instance, a health clinic may acquire cloud storage services with a maintenance contract. In such cases, the service provider's IT personnel has access to the clinic's information, which is to be kept confidential by law.

Ideally, IT workers should have access only to resources they actually need, while any mechanism for isolating them

*Part of this work was done during an internship at IBM Research Haifa.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132752

from other resources should not interfere with their work. The challenge in creating appropriate isolation for system administrators lies in the coarse-grained permission model employed in commodity operating systems like Linux. Better defense against privileged insider attacks therefore requires finer-grained control over privileges.

We introduce *WatchIT* which implements such fine-grained control in the context of containers, a virtualization method already widely deployed in cloud systems and development environments today. A container envelops a set of processes and gives them an isolated view of the operating environment. On the face of it, the isolation that containers provide goes against the very nature of system administration – indeed, IT personnel get permissions for a reason.

Perhaps counter-intuitively, we propose in this work to exploit containers for system administration. Our idea is to put holes in the isolation of traditional containers in order to create a middle-ground that is useful for administrators and yet controlled. We introduce the abstraction of a *perforated container* – a container that may share system resources with the host, under logging and monitoring. This perforated container is essentially a sandbox that constrains IT personnel's view of the system. It provides a way to monitor their actions, yet preserves their superuser privileges within specified boundaries.

The solution we devise is based on three principles: (1) isolating the system administrator from resources that are expected to be irrelevant (using a perforated container abstraction), (2) enabling her to perform actions beyond the isolation boundaries under monitoring and logging, and (3) optionally monitoring the allowed operations executed inside the perforated container. To support the second, we introduce a *permission broker* – a software service that can change the boundaries of the perforated container and perform operations on the administrator's behalf.

Current container software such as Docker [30] and others [3, 4, 33] does not provide the required flexibility for configuring and deploying containers for administration. For example, a container cannot share the host's view of the filesystem or provide access to certain file types while excluding others.

We implement *ContainIT*, dedicated container software for building perforated containers. ContainIT can share the host's filesystem view as well as other namespaces (e.g., processes or network). It can further log and monitor network activity and filesystem operations. Moreover, it allows for login of privileged users but can block access to specific files even if the contained administrator can see that they exist. Via the permission broker, a ContainIT perforated container can obtain additional filesystem and network views on-demand, without restarting. Furthermore, ContainIT is able to utilize a new namespace we introduce, *exclusion namespace*, which excludes access to parts of the filesystem for the processes it contains, even if its associated container shares the host's filesystem table.

We complement our container software with a framework that orchestrates our approach, and is deployed within the trusted computing base of each organizational machine. The framework includes a permission broker and a software system that can receive a free-text IT problem description, classify it, and deploy a corresponding perforated container at the target machines.

To illustrate the feasibility of our approach, we present a case study conducted in the IT department of IBM Research, Israel. We used the department's database, which includes tens of thousands of user reported *tickets* and dozens of maintenance scripts; we further collected hundreds of IT tickets in real time during a three-month evaluation period. The case study shows that IT tasks can indeed be divided into categories and classified using common classification algorithms. By adjusting a perforated container for each ticket class, we were able to accurately capture the needs of 92% of the 398 cases observed during the evaluation period. We used the permission broker to complete the remaining 8% of the cases. Our results showed that our solution denied full filesystem view in 62% of the cases, and isolated the network view in 98% of the cases, while all filesystem operations and network traffic were monitored. Overall, we compartmentalized IT personnel from resources that were indeed proved to be irrelevant to the ticket solving in all of the observed cases, thus reducing the IT attack surface. Following the success of the case study, IBM is working to integrate a WatchIT-based solution in its products whose consumers are required by law to preserve their clients' information confidentiality.

Our contributions in this paper are as follows:

(1) We devise a novel container-based approach to protect organizations from their system administrators.
(2) We provide a proof-of-concept implementation of our approach, which includes new container software, a framework for deploying perforated containers, and required changes to the Linux kernel.
(3) We present a real-world case study, based on the IT department database and workflow of IBM Research, Israel.

The rest of this paper is organized as follows: We state our threat model and detail the current state-of-the-practice in Sections 2 and 3, respectively. We describe the main concepts of our approach in Section 4, and give a detailed overview of WatchIT and its implementation in Section 5. Next, in Section 6 we discuss ways to circumvent WatchIT and the measures we take to mitigate them. In Section 7 we present a case study, performed using a real IT department, through

which we evaluate our approach. Finally, in Sections 8 and 9 we discuss related work and conclude.

2 TCB AND THREAT MODEL

We utilize standard techniques and build upon a Trusted Computing Base (TCB) to boot the WatchIT framework into a trusted initial state. We consider a TCB that provides a secure environment, which includes the system hardware, the operating system with all its built-in security controls, system drivers, system services, and WatchIT components. An example commodity product that provides such TCB functionality is BitLocker [25], which validates the integrity of boot and system files before decrypting a protected volume.

Given such a TCB, our threat model is a single rogue IT employee who wishes to access confidential data, directly or indirectly by installing malicious software, that is considered irrelevant for the specific ticket context. We assume a workflow in which clients report trouble tickets which are later assigned to specific IT personnel, based on expertise or required permissions. The assignment creates a certificate that allows the designated system administrator to login into a perforated container deployed on the target machine for a limited time. System administrators do not otherwise have access to the machines in question, and they cannot create trouble tickets on their own initiative.

Once an administrator is logged into a perforated container, she can modify all system resources to which she is exposed, as long as she does not change the TCB. Our system blocks all actions that may change the TCB signature; thus, an IT person cannot change the OS kernel, install unauthorized drivers or kernel modules, or install non-certified services. These special actions require escalation, provided by the permission broker, and thus allow WatchIT to audit the change and make sure it is signed by the organizational policy system. Note that these assumptions are supported by the case study presented in Section 7, as less than 1% of the tickets we encountered involved driver updates.

3 STATE-OF-THE-PRACTICE

In Section 3.1 we provide the motivation for our work by discussing the state-of-the-practice in IT workflow and pinpointing the danger that IT personnel poses to organizational information security via three use-cases. Next, in Section 3.2, we give essential background on container technology.

3.1 IT Vulnerability

IT personnel usually have superuser privileges on the machines in their responsibility domain. Indeed, superuser permissions are essential for system administration. However, due to the coarse-grained permission granularity in traditional Linux systems, an IT person practically has access to any file or configuration stored in these machines. We identify three main scenarios in which IT employees can take advantage of their permissions to steal classified data.

Daily IT Support. IT workflow usually consists of three main steps: (1) The end user fills out a ticket that describes the request, usually in free text. (2) The IT department receives the ticket and dispatches it to an appropriate IT specialist. (3) The IT specialist gains access to the computer in question, usually remotely, and attends to the request under superuser privileges on the target machine.

The last step constitutes a major security breach. While attending to the ticket, the IT specialist has unsupervised access to all the resources of the target machine, even though most of these are irrelevant to the ticket at hand. This can be exploited to steal classified commercial data, copy personal information, or install malware on the end-user's computer.

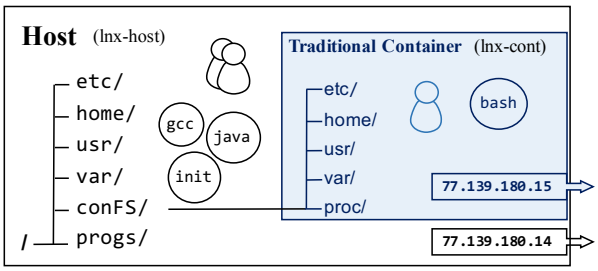
Third-Party Support. Such threats come not only from the organizational IT department, but also from external service providers, e.g., a company that provides storage solutions for various customers such as banking systems, health companies, etc. Beyond providing the product or service itself, the company also remotely provides support, which requires superuser privileges on the customer's machines. Furthermore, the service provider is logged into the customer's organizational network, and is exposed to data on the target machine as well as data on other machines in the network. Thus, the service provider might be exposed to data that must remain confidential by law. For example, this may include credit card information that must comply to the PCI data security standard and medical records subject to HIPPA compliance. Moreover, if the service provider is not an employee of the storage company, but a third-party IT contractor, the leakage can be very difficult to trace.

Automatic Management Tools. IT departments usually employ automatic management tools, such as Chef [14] and Puppet [35], in order to perform various administration and maintenance tasks. These tools operate using dedicated scripts, which are written by the IT personnel and designed to verify system configurations and properties. These scripts may run periodically or be executed manually as part of handling specific IT requests.

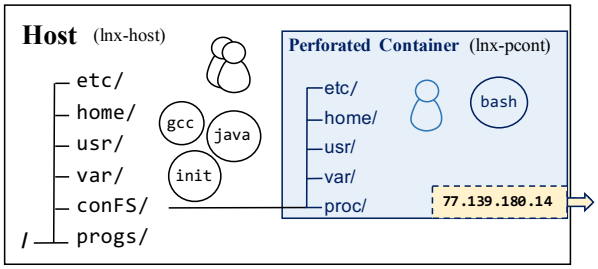
Automatic management tools run with superuser privileges, without any sandbox to monitor the script's operations. An IT insider can tamper with these scripts, causing them to install malware or leak classified data, thus compromising many machines at once.

3.2 Containers

Containers offer an operating system level virtualization technique for running multiple isolated systems (containers)



(a) Traditional container example.



(b) Perforated container example.

Figure 1: Traditional vs. perforated container example. A traditional container is associated with a different namespace of each available type, while a perforated container may share a namespace with the host (the network namespace in this example).

on a host running a single Linux kernel. Virtually, a container is a group of processes, isolated in various resources from the host system on which they run. The resource isolation in modern Linux-based containers is provided by the *namespaces* mechanism [28]; when a process is associated with a different namespace from the host, it has a different view of the corresponding resource. There are six namespaces in Linux; these provide per-container views of the following resources (see Figure 1):

(1) UNIX Time Sharing (UTS) – affects the view of the host-name. For example, in Figure 1a, processes inside the container read the host-name of the machine that they run as `lnx-cont`.
(2) Mount namespace (MNT) – provides different views of the mounted filesystem table to different processes running on the same Linux host. For example, in Figure 1a, we see a container that is able to see only the files mounted under `/conFS/` directory.
(3) Network namespace (NET) – controls the network view of the contained processes. Processes that belong to the same NET share routing tables, firewall rules, and network devices (represented by IP arrow boxes in Figure 1a).
(4) Process ID namespace (PID) – provides different views of the processes in the system. Processes that are associated with a given PID namespace can only see each other and processes in child PID namespaces.
(5) Inter-process communication (IPC) – affects IPC objects that are identified by mechanisms other than filesystem pathnames, such as shared memory.
(6) User ID namespace (UID) – provides different views of the IDs of the users in the system. This enables mapping of contained users to host users, thus giving corresponding permissions to contained users on resources in their view.

As demonstrated in Figure 1a, a traditional Linux container associates the processes that it contains with a new namespace of each available type.

Overall, a container offers an environment with the salient benefits one gets from a VM, but without the overhead that comes with running a separate kernel and simulating the hardware. As a result, the deployment time of containers is a matter of seconds (if all the required files are available locally), and significantly shorter than virtual machines.

4 PERFORATED CONTAINERS

Perforated Containers. Protecting a system from IT insider threats goes through fine-grained control over privileges along with monitoring. To satisfy these demands, we exploit the isolation properties of traditional Linux containers, and contrary to their nature, puncture this isolation – thus introducing *perforated containers*.

Perforated containers allow us to associate a process with only a subset of the available namespaces, thus allow it to share the remaining resources with the host. For example, Figure 1b illustrates a container that associates the processes it contains with all the available namespaces except for the network namespace, which is shared with the host. Thus, processes running inside the perforated container can only see the files under `/conFS/`, read the host-name as `lnx-pcont`, and so forth; this is similar to the traditional case. However, containerized processes have the same network view as non-containerized ones; namely, the perforated container and the host share routing tables, firewall rules, and network devices. Such a perforated container might be useful for repairing connectivity problems.

In some cases, the perforation might be too permissive. To this end we would like to inspect the operations done from within the perforated container, and the information that flows through these holes. Therefore, alongside the resource sharing, the perforated container's boundaries should be monitored; thus turning it into a sandbox environment. Such monitoring allows us to perform network packet inspection for a container's network traffic, filter filesystem accesses by content or file type, log various operations, and more.

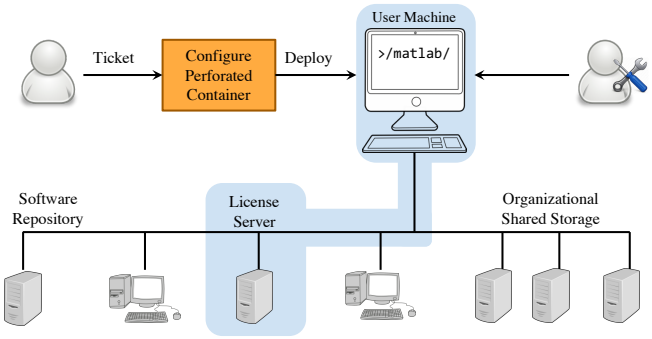


Figure 2: Example of deploying a perforated container designed to handle Matlab licensing problems.

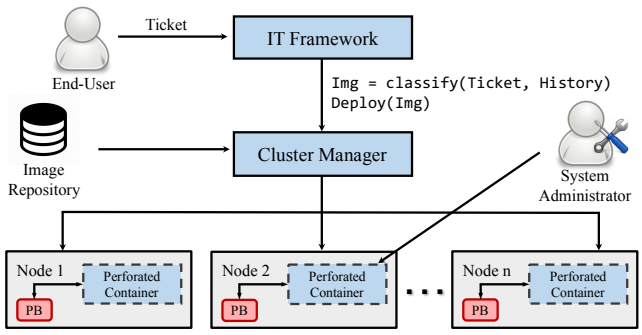


Figure 3: WatchIT architecture overview. PB is the permission broker.

Custom Tailoring. Our proposed approach is to deploy perforated containers as a sandbox mechanism for IT operations. We adopt the Multics [40] principle of least privilege, and thus for each IT request, we sew a custom-made perforated container that allows access only to resources that are relevant to that request. We follow another Multics security principle and base the protection on permission rather than exclusion. For example, as illustrated in Figure 2, if the request requires fixing an expired Matlab license, the deployed custom-made perforated container should have filesystem access only to the Matlab directory, and its network view should include only the organizational license server. The view of other filesystem parts, as well as other network nodes, and even other system processes, should be unavailable. We configure the perforated container boundaries, as explained next, by processing the user request and predicting the maximal isolation under which the IT requests can be handled.

Bypassing and Monitoring. By perforating the isolation provided by traditional containers, we create a supervised, controlled, and flexible sandbox environment for each IT request. Nevertheless, our prediction cannot be perfect, and there are bound to be cases in which the configured isolation is too restrictive, and not sufficient for completing the designated IT request. To this end, we propose a *permission broker*. The permission broker is a software service that runs on the host and provides two complementary mechanisms. First, it allows administrators to bypass the container boundaries by executing commands on behalf of the perforated container. Second, it can grant the perforated container additional permissions (system views), thus effectively expanding its boundaries. Requests sent to the permission broker can be denied or approved, according to a predefined policy. Moreover, all such requests are logged for future analysis, thus enabling monitoring, better adjustment of container limits for future tickets, and improved investigation capabilities in case of security breach.

Benefits. Exploiting containers for administration has the following benefits: First, our approach supports the creation of various perforated containers, each with a different isolation set. This allows for fitting a container for each IT mission, in a fine-grained granularity. Second, a superuser inside the container may indeed have the privileged capabilities it needs. Nevertheless, these privileges are effective only on the resources that are visible to the container. Third, containers can be deployed within seconds, given that all needed files exist on the target machine; this is valuable for administration tasks, which usually should be done in a timely manner. Fourth, namespaces are supported in all the latest Linux distributions. In addition, as explained later, our approach enables efficient monitoring and auditing of IT, including filesystem operations and network traffic.

5 WATCHIT

In this section we present *WatchIT*, and dive into its implementation. In Section 5.1 we give an overview of the system structure. Next, in Section 5.2 we present our perforated container software, ContainIT, and discuss its architecture. We then present two mechanisms that ContainIT uses: ITFS (Section 5.3) for inspecting IT file operations and the permission broker (Section 5.4). Next, we discuss in Section 5.5 our technique for changing a running container filesystem view. Finally, in Section 5.6 we propose a new namespace for the Linux kernel, motivated by this work.

5.1 Architecture Overview

The architecture of our proposed system is depicted in Figure 3. The workflow 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 describe software problems, networking issues, expired licenses, and so on.

The system is pre-configured with a number of ticket *classes*, each associated with a dedicated perforated container

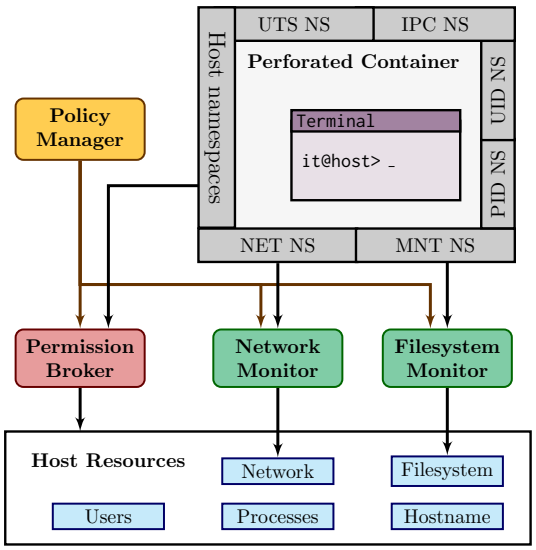


Figure 4: ContainIT software architecture. NS stands for namespace.

encapsulating the privileges required in order to attend to tickets of this class (via namespace settings, installed software, and monitoring configuration). Like the Docker architecture [30], the various container images and configurations are held in a dedicated image repository for quick deployment. New tickets are submitted to an organizational *IT framework*. The framework analyzes the tickets and classifies each ticket to one of the predefined ticket classes, based on history and ticket parameters. The classification is performed automatically, and reviewed by the user or a supervisor.

Upon classifying the ticket, the framework asks the cluster manager to deploy the corresponding perforated container image on the target machines. Following the deployment, IT personnel can log into the deployed containers in the target machines and attend to the ticket. As proposed in previous works [45], connecting to the deployed perforated containers is enabled via a temporary certificate, which is revoked once the ticket time expires. 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 to which the container is not exposed.

5.2 ContainIT

We now describe *ContainIT*, our software for deploying perforated containers. Commodity container software (such as Docker or LXC) does not support many of the features we require, including custom made perforated containers, monitoring container filesystem operations and network traffic, flexible on-line file-sharing, and more. Therefore, we build ContainIT, our own container software, which supports these features. ContainIT is written in C.

Figure 4 presents the software architecture of ContainIT. First, the boundaries of a perforated container are determined by its attributed namespaces (abbreviated NS). As explained in Section 4, not all available namespaces must be used, and access to resources governed by missing namespaces is unconstrained. For example, if the perforated container shares the host's PID namespace (because this namespace is excluded), then the IT person may freely communicate with processes on the host.

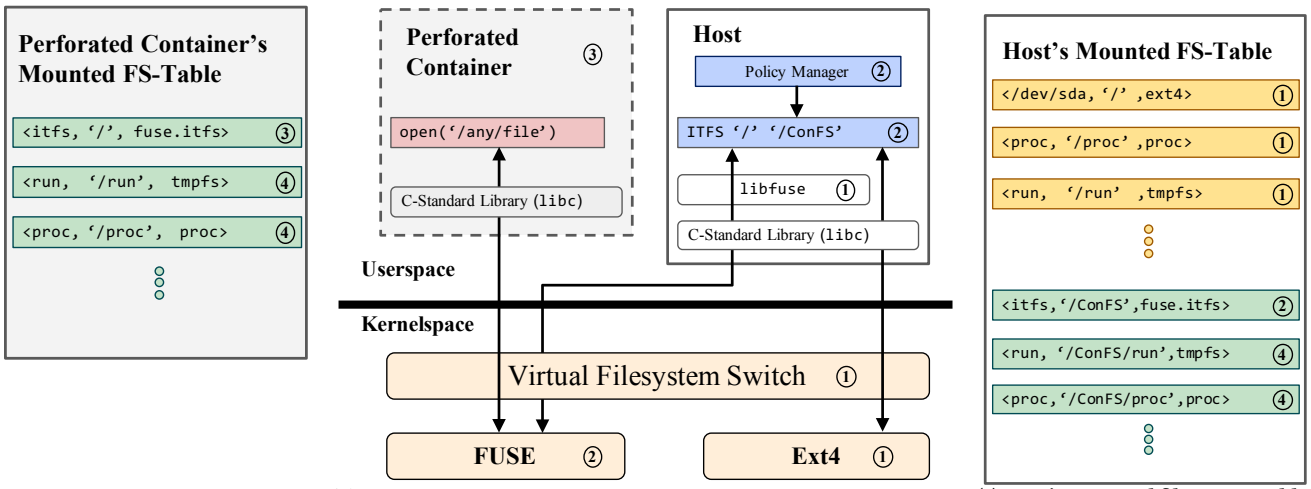
Second, accesses to network and filesystem resources through the corresponding namespace are monitored (green boxes). For example, network traffic going through network devices associated with the perforated container's NET namespace is tapped, analyzed, and can be blocked if necessary. To implement this feature, we make use of existing sniffing mechanisms [32, 38]. To monitor filesystem operations, we build ITFS – a FUSE-based filesystem that traps file system calls, allowing the inspection, blocking, and logging of file operations, as detailed in Section 5.3. Conceptually, this monitoring approach may be applied to additional namespaces, but we did not find a need for it.

Finally, for cases in which a contained user needs to access the host's resources, she can contact the permission broker (pink box) – a software service running on the host with unlimited access to the host's namespaces. Requests sent to the permission broker are logged, inspected, and can be denied or accepted. The policies of the permission broker and the namespace monitors are dictated by the *policy manager* (yellow box).

This software architecture makes our container software highly configurable. Examples include a perforated container that shares the host's root filesystem but cannot access document files; a perforated container that cannot see the host's filesystem, yet is able to see and kill the running processes on the host; a perforated container that can be accessed via SSH, but has no access to the world-wide web.

5.3 ITFS – Filesystem Monitor

We now describe the mechanisms we provide for monitoring file accesses by IT personnel as well as for sharing the host's root filesystem with a perforated container. For these purposes we build *IT File-System (ITFS)*, a FUSE-based [37] monitoring filesystem that can deny or log accesses to the underlying filesystem according to configurable rules. Note that current container software (Docker, LXC) does not provide the ability to deploy a container that has the same view of the filesystem as the host's. Since this is an imperative feature in administration, we provide it here, alongside the monitoring mechanism.



(a) Perforated container's mounted filesystem table. **(b) FUSE software architecture. Arrows represent the flow of file system calls.** **(c) Host's mounted filesystem table.**

Figure 5: Using ITFS in order to monitor a perforated container's filesystem operations. The perforated container's mount point is `/ConFS/`. The run-time flow is depicted in Figure 5b.

We build our ITFS solution in two steps. First, we mount the host's root filesystem at a dedicated mountpoint, while trapping all the file system calls targeted at that mountpoint. Second, we `chroot()` the perforated container to the dedicated mountpoint. Namely, we change the filesystem view of the container by making it see the dedicated mountpoint as the root of the system. Assuming that the perforated container has a different MNT namespace than the host, this mechanism ensures that all filesystem operations performed within the perforated container are monitored.

Figure 5 depicts how our mechanism inspects filesystem accesses initiated from within a perforated container. (1) In the initial state, the host only has the Ext4 filesystem registered in its VFS, while its mounted filesystem table (Figure 5c) only contains the yellow entries. (2) Next, we prepare the ground for deploying the filesystem-monitored perforated container by invoking ITFS on the host and mounting the root of the host's filesystem on a dedicated mount point (the directory `/ConFS/` in this example). This invocation adds an entry to the mounted filesystem table of the host (first green entry the in Figure 5c), and forwards all future file system calls targeted at this mountpoint to the FUSE kernel module, which registers a new filesystem in VFS.

(3) When deploying the perforated container, we configure its filesystem to be rooted at `/ConFS/`. Since the perforated container has a separate MNT namespace, it is only aware of the ITFS filesystem mounted on its root address. This is the first entry of the perforated container's mounted filesystem table shown in Figure 5a. (4) Additional future filesystems mounted in the perforated container's namespace scope (`run` and `proc` green entries in Figures 5a and 5c) are rooted at `/ConFS/`. As shown in the figure, all the entries of the perforated container's filesystem table have counterparts in the host's filesystem table.

Once deployed, every file system call issued from the perforated container is forwarded through VFS, by the FUSE kernel module, to the libfuse library running on the host. Hence, as illustrated by the arrows in Figure 5b, when a containerized application calls `open()` (pink box), the call is forwarded to ITFS, which invokes the corresponding callback defined in its policy (blue boxes) for the `open()` operation. The callback may take *any action*, and return the desired data in a supplied buffer. For instance, it can read the file from the underlying filesystem (such as Ext4), detect its type according to its signature, and deny access if the file is a picture or a document. Alternatively, it can allow access to the file, but log this access for later analysis. Finally, the return value is propagated back by libfuse, through the kernel, to the application that issued the `open()` inside the container.

We further provide an API for configuring the monitoring rules and actions. The API supports forbidding and/or logging access to a file according to its signature or extension. This allows, for example, control over access to documents – files that are usually irrelevant to IT work. In addition, ITFS exposes an API for integrating user-supplied detection rules, as scripts or programs, so that each organization can create customized file filtering.

Under the approach described in this section, the user logged in to the container inherits the privileges of the user

that invokes the ITFS on the host; this happens in all FUSE-based filesystems by design. Thus, if ITFS is mounted with superuser privileges, the user inside the container also has superuser privileges for all the files that are exposed to the container through ITFS. Indeed, this is the desired behavior for administration tasks.

Note that the presented approach is possible only if the perforated container has a different MNT namespace from the host, which is usually the case. As shown in the case study in Section 7, none of the examined tickets required sharing the MNT namespace of the host. Under this condition, ITFS can inspect the file accesses of the contained IT person, without allowing her to bypass this mechanism, even if the perforated container is admitted to the host's root filesystem itself. For cases in which the administrator does need access to the host's entire MNT namespace, we provide the XCL namespace, as discussed in Section 5.6.

5.4 Permission Broker

On some occasions, the permissions assigned to a container may be insufficient for handling the ticket. To address such cases, we augment the system with a *permission broker*. The permission broker is a software service that runs on the host and can grant a running container additional permissions, change its filesystem and network view, 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 wishes to see the list of running processes in the system (from which she is compartmentalized), she can submit a request to the permission broker, as shown in Figure 6. In the example, executing "`ps -a`" from the container shows only processes that belong to the container's PID namespace. By requesting the command from the permission broker (using the `PB` prefix), the contained user may see the host's processes.

The permission broker grants a request if it follows the security policy corresponding to the specific ticket class and IT specialist, and can refuse otherwise. Either way, these requests are logged in real-time to a secure append-only storage device, and can be monitored and analyzed later for anomaly detection. As a consequence, our permission broker logs only IT activities that diverge from the predefined isolation, which captures the expected behavior for handling a specific ticket. Hence, the permission broker's log is sufficiently succinct to be inspected and analyzed for anomaly detection [13, 15, 39, 48], where one of the major challenges is handling enormous amounts of data. Moreover, if certain permissions are repeatedly requested, they can be added to the ticket class's perforated container, thus further reducing

```
root@ITContainer:/home/itsupport# ps -a
  PID TTY          TIME  CMD
    1 pts/4    00:00:00  containIT
   32 pts/4    00:00:00  bash
   71 pts/4    00:00:00  testscript
   73 pts/4    00:00:00  ps
root@ITContainer:/home/itsupport# PB ps -a
  PID TTY          TIME  CMD
 1023 pts/14   00:00:00  PermissionBroker
 1075 pts/4    00:00:00  sudo
 1077 pts/4    00:00:00  ContainIT
 1080 pts/4    00:00:00  itfs
 1081 pts/17   00:00:00  snort
 1139 pts/4    00:00:00  bash
 1272 pts/18   00:00:00  testscript
 1276 pts/19   00:00:00  ps
root@ITContainer:/home/itsupport#
```

Figure 6: Example of using the permission broker.

the amount of gathered data, and in turn facilitating future data analysis.

We implement the permission broker in Python, using a client-server architecture, where the server side is installed on the host and the client side is invoked from the container. The communication goes through the tcp/ip stack, and we use Google's Protocol Buffers and gRPC [19] for serializing and streaming the data. In order to prevent regular users from contacting the permission broker, we configure the permission broker client to accept only requests from privileged users. If one wishes to further secure the communication between the perforated container and the permission broker, one can employ SSL.

In order to change the system view of the deployed perforated container, the permission broker performs operations on the routing tables and firewall rules of the container's namespaces and uses the `nsenter` tool as detailed next in Section 5.5.

5.5 Online File Sharing

Commodity container software such as Docker provides means for mapping directories from the host filesystem into a non-root directory in the container filesystem. However, such mapping must be requested at launch-time; once the container has been deployed, there is no support for exposing additional host directories to the container.

In WatchIT, on the other hand, we would like to allow the permission broker to map additional directories on-the-fly; namely, expose additional directories to the perforated container while it is running, without requiring it to restart.

Hence, we equip ContainIT with the ability to perform on-the-fly file sharing, while ensuring that subsequent contained accesses to newly shared files are monitored by our ITFS mechanism. Mounting additional directories into the container filesystem is not trivial. Due to the namespace hierarchy, mount operations on the host are not visible in

the container; hence, the mounting should be executed from within the perforated container. On the other hand, the perforated container is not aware of the files from which it is isolated; hence the operation cannot be purely executed from within the container. Our solution employs nsenter [8], a Linux tool for entering the namespaces of a given process and executing programs within them.

The full process of adding a new volume to a running container begins by issuing a request from the container to the permission broker. Next, the permission broker logs the request and turns to execute it under superuser privileges. The implementation of the feature itself is comprised of three main stages: (1) extracting the full real path to the host directory and device ID on whose filesystem the directory resides. (2) using nsenter in order to infiltrate the namespaces of the running perforated container; and (3) creating an ITFS bind mount to the host directory in the target path from within the container's namespace.

Since we create an independent ITFS bind mount, accesses to the newly mounted filesystem are supervised by ITFS, but can have different rules. Moreover, even if one chooses not to employ ITFS monitoring on the originally deployed perforated container, one can still employ it only on the newly mounted files.

Finally, we note that our online file sharing technique does not constitute a security breach in the Linux namespace mechanism. It is possible only because it requires superuser privileges on the host. In our case, these privileges are provided by the permission broker.

5.6 Exclusion Namespace

The mechanism described in Section 5.3 for sharing the host's underlying filesystem with the perforated container under supervision and monitoring is only possible when the container has a different MNT namespace from the host. However, it may be the case that the administrator needs to access the host's entire MNT namespace, for example, if the system administrator needs to handle problems with the filesystem itself or mount more filesystems on the machine. In these cases, the perforated container and the host have the same view of the mounted filesystem table, and there is no guarantee that each filesystem operation of the containerized superuser is monitored.

To solve this problem, we introduce a new namespace to the Linux kernel – *exclusion (XCL) namespace*. The XCL namespace has a table of excluded directories – filesystem sub-trees that cannot be accessed by processes that belong to that namespace, disregarding the user privileges. Thus, even if a containerized superuser has access to the underlying filesystem, she will not be able to access the parts of the filesystem that are specified in the exclusion table.

We implementat the XCL namespace in version 4.6.3 of the Linux kernel. Following Linux conventions, one can associate a newly created process with a new XCL namespace instance by executing the *clone()* system-call with the flag CLONE_XCL. Entries can be added to and removed from the excluded directory table using dedicated system-calls. A newly created namespace instance inherits its parent's exclusion table.

6 THREAT ANALYSIS

We now discuss the threats facing WatchIT, which we summarize in Table 1, and the measures we take in order to neutralize them. In Section 6.1 we analyze methods for breaching WatchIT protection from within a perforated container using technical skills. Section 6.2 discusses ways to circumvent the WatchIT system.

6.1 WatchIT Software Security

Keeping Perforated Container Boundaries Safe. Our perforated container is created by punching holes in the isolation provided by traditional containers. Moreover, a perforated container may map a contained user to a privileged one on the host, since it may be required to perform operations like service restarts or system reboots. The holes in perforated containers alongside privileged permissions might be abused to escape the perforated container boundaries.

There are four known techniques to escape a chroot()-ed environment (including containers) [44]. The most common one requires root privileges, and issues a new chroot() command in order to escape the current one. The second technique is based on communicating with a process outside the container, changing its code using ptrace(), and turning it into a bind shell. The third builds on creating raw disk devices and mounting existing filesystems on them. The fourth creates /dev/mem or /dev/kmem devices, thus tapping and modifying system and kernel memory.

Using the Linux capabilities feature [2], we deprive contained users of three capabilities that are essential for employing the first three aforementioned bypassing techniques (Attacks 1-3 in Table 1), and are also rarely needed in IT work. These are: chroot(), ptrace(), and mknod(). We further implement a new capability and employ it to block a contained user from opening /dev/mem and /dev/kmem (Attack 4).

Previous work [11] presented nine ways to escape chroot()-ed environments by exploiting privileged permissions, which use similar capabilities as the attacks described in [44]. We tested each of them in containers deprived of the capabilities mentioned above, and verified that they cannot escape the perforated container.

ID	Attack	Defense	Weaknesses
1	Escape perforated container boundaries	Block chroot() capability	
2	Bind shell	Block ptrace() capability	IT cannot perform debugging
3	Raw disk mounting	Block mknod() capability	IT cannot create special files
4	Memory tapping	Block access to /dev/mem & /dev/kmem	
5	Tampering with WatchIT software	(1) Include WatchIT in TCB (2) Block access to WatchIT files	
6	Tampering with log files	(1) Replication (2) Block access by ITFS	
7	Kill WatchIT component	Other components exit and terminate session	
8	Encrypt and exfiltrate	(1) Blocks filesystem access (ITFS) to unencrypted files (2) Sniff or block suspicious network traffic	Requires ITFS and network sniffer rules
9	Fake tickets	IT personnel cannot create trouble tickets	Collusion with non-admin insider
10	Ticket stringing	(1) Permission-based ticket assignment (2) Imposing hard constraints on all perforated containers	
11	Malware installation	(1) Website whitelisting (2) Monitor incoming network traffic	Watering hole attacks, phishing, etc.

Table 1: Possible attacks on WatchIT, the measures we take to neutralize them alongside their main weaknesses.

Protecting WatchIT Software. We prevent the contained administrator from tampering with our sandbox mechanism in two ways (Attack 5). First, we include WatchIT software (including the permission broker, policy manager, and ITFS software) in our TCB, so the system will not boot if any of its components have been tampered with. Second, we use ITFS to block accesses to all WatchIT files; this way, we deny access to these files even if the contained user can view them. Furthermore, in order to prevent the log files from being compromised, they can be replicated on a remote append-only storage (Attack 6). Finally, we design ContainIT to terminate the session if any of its peer processes (e.g., permission broker service) are killed (Attack 7).

Protecting Files within a View. Network sniffer software [32, 38] mostly relies on detecting the signatures of files sent over the network. Hence, a common attack would be to tamper with the victim files or conceal their content by encryption and send them over the network (Attack 8). In order to protect from such an attack, the ITFS blocks the actual access to files that are defined as classified by the policy manager. Since one cannot tamper with a file without reading it, this prevents exfiltration.

6.2 Circumventing WatchIT

Collusion. WatchIT is designed to protect against a single adversarial system administrator and is vulnerable to collusion between a user and an IT person (Attack 9). That said, we note that the state-of-the-practice is that administrators can act solo, so making collusion necessary significantly raises the bar for attack; in particular, when IT is managed by a third party (possibly in a different country). Furthermore, filesystem operations performed from within a perforated container are monitored by ITFS and since all tickets are recorded, collusion leaves a trail.

Ticket Stringing. If an administrator is assigned to handle multiple ticket types, a possible attack would be to sequentially string tickets, thus concatenating system views and expanding her effective permissions (Attack 10). An effective solution against such threats is to impose hard constraints on all perforated containers. For example, as presented in the case study in the next section, imposing an ITFS policy (e.g., forbidding access to documents) and network sniffer rules (e.g., disallowing transfer of encrypted files) on all perforated container classes, prevents data leakage even in the face of stringing. Moreover, in large organizations, WatchIT can be protected from such threats by assigning to each IT person only tickets of the same class.

Group-Targeted Attacks. Organizations are usually exposed to group-targeted attacks, like phishing and watering hole (Attack 11). WatchIT is not designed to provide protection from such threats. Thus, for example, watering hole attacks on whitelisted sites that are used for software downloading (e.g., Eclipse) remain dangerous. Nevertheless, they affect regular users as well as administrators. Moreover, if WatchIT is not in use, administrators accessing these sites allow the downloaded malware broader system view and permissions than when WatchIT is used.

Finally, we note that WatchIT supervisors who create the perforated containers and define their system view remain omnipotent. However, we narrow the trust group from many administrators, possibly including third-party contractors, to very few.

7 CASE STUDY

This section presents a case study of applying WatchIT on the IT database of IBM Research in Israel. The studied IT department consists of about 30 system administrators, among them 7 Linux specialists, supporting around 600 technical users. We begin in Section 7.1 with a thorough ticket analysis, clustering, perforated container tailoring, and testing on

Topic T-1	Topic T-2	Topic T-3	Topic T-4	Topic T-5	Topic T-6	Topic T-7	Topic T-8	Topic T-9	Topic T-10
license	password	file	connect	work	install	VM	access	connect	space
matlab	user	< Shared Storage >	< IP >	time	< Server >	< VM >	user	< Server >	project
error	connect	access	port	machine	version	GB	add	SSH	< Shared Storage >
DB2	account	SVN	server	slow	< OS >	IP	group	respond	GB
toolbox	login	directory	network	stuck	upgrade	disk	team	VNC	increase
message	locked	git	IP	reboot	< Application >	kvm	permission	LSF	quota

Table 2: Results of running 10-topic LDA on our data set. For each topic we present six of its top 20 words.

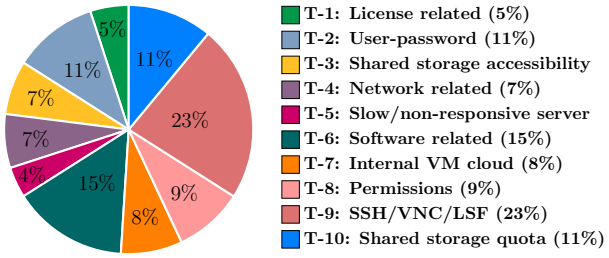


Figure 7: Category assignment and distribution.

real-world workload. Next, in Section 7.2, we audit scripts used by the same IT department (e.g. Chef, Puppet) and show that perforated containers can be suited to protect from tampered scripts. Finally, in Section 7.3 we evaluate ITFS performance and show that it can indeed be used as a filesystem for administration tasks.

7.1 IT Tickets

We apply the WatchIT approach to tickets of a real-world IT department. We employ statistical tools to cluster the tickets into classes and validate the resulting division with IT personnel. Next, we custom-tailor perforated containers and test their compatibility on new tickets collected during a trial period.

7.1.1 Ticket Clustering. For our case study, we analyze an IT database containing about 66,000 tickets, collected during the years 2009 through 2016. Tickets are written in free text by end-users and are manually sent by the IT department to the corresponding IT specialist. The actions taken by IT personnel to handle each of these tickets are not included in the database. 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 who specialize in Linux issues. This leaves us with a corpus size of around 17,000 tickets.

To group the tickets into categories, we use topic modeling [9]; such algorithms take a corpus and group the words across it into topics. Before performing topic modeling, we pre-process the corpus by applying word stemming, stop word removal, deletion of common words that do not add information (like 'hello' and 'please'), and obfuscation of confidential information such as server names, addresses, project names, etc. We then use Latent Dirichlet Allocation (LDA) [10] to process the data and group it into a specified number of topics. We run LDA with 7 to 14 topics and choose the most appropriate result, which in our case consisted of ten topics.

Partial results of the ten-topic LDA analysis appear in Table 2. LDA represents a topic as a distribution over words; thus, the full results of LDA for each topic include 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 of finding that word in a text on the corresponding topic. For illustration purposes, Table 2 shows only six representative words for each topic, taken from the top-twenty words of each list. Surrounding angle-brackets represent names or addresses; e.g., <IP> stands for any IP address.

One can infer from the results the main categories that the studied IT department deals with. The category distribution appears in Figure 7. For example, topic T-1 constitutes 5% of our tickets and refers to license problems, which are usually associated with Matlab, Matlab toolboxes, and database software; topic T-5 refers to slow or non-responsive servers; and topic T-6 is associated with software related requests, as among its top words we can find application names like eclipse, gcc, and hadoop, and words like "install", "upgrade", "version", "package", and "plugin".

To verify our topic choices, we interview the IT personnel without exposing them to the LDA results. Our interviews with the IT personnel yielded 13 main categories, which are all included in the LDA results, although with some modifications: (1) IT personnel treated SSH, VNC, and Load Sharing Facility (LSF, a batch job execution environment) issues as different categories, while LDA groups them into one (Topic T-9). This happens since the descriptions of such problems often use the same words. (2) IT personnel distinguish between shared storage accessibility problems to SVN and git issues, while LDA mixes them into one. Again, this happens for the same reason – organizational SVN and git repositories reside on the organizational shared storage, and all related IT issues are associated with creating and managing accessibility to these repositories; hence the same words are used to describe these issues.

	Process Management Permission Set	Filesystem Access			Network Access						
		Home Directory	/etc/	Root Directory	License Server	Batch Server	Shared Storage	Target Machine	Software Repository	Whitelisted Websites	Network Namespace
T-1: License related		X			X						
T-2: User / password			X								
T-3: Storage accessibility	X	X					X				
T-4: Network related	X		X		-	-	-	-	-	-	X
T-5: Slow server	X	-	-	X							
T-6: Software related	X	-	-	X					X	X	
T-7: Internal VM cloud			X								
T-8: Permissions		-	-	X							
T-9: SSH/VNC/LSF	X	X	X			X		X			
T-10: Storage quota			X				X				
T-11: Other											

Table 3: Permission and isolation per each container type. We denote by "X" explicitly included resources, and by "-" resources that are implicitly included due to the inclusion of another resource.

However, as presented in the next section, in both of these cases, the mixed categories also share the required permissions for handling their associated tickets. Indeed, SSH, VNC and LSF issues always necessitate a connection to a remote server, and involve changing local configuration files. Similarly, SVN and git repositories always reside in the shared organizational storage and are associated with accessibility issues. We conclude that the LDA output is accurate and sufficient for mapping tickets to the needed permissions.

7.1.2 Permission Assignment. Given the above classification, we consult with the IT personnel and build ten perforated containers, which differ in their permissions and resource isolation. The isolation and permissions for each ticket class appear in Table 3; alongside the isolation, filesystem accesses are monitored by ITFS and network traffic is sniffed by IDS software.

For example, the T-1 perforated container for attending to licensing problems can modify the home directory of the user and connect to the organizational license server (a server responsible for company license management, maintained by the IT). Nevertheless, it is compartmentalized from the rest of the filesystem and other nodes in the network.

The process management permission set includes the ability to (1) see and kill the host's running processes, (2) restart host's system services, and (3) reboot the machine. Whereas T-1 (the licensing container) is isolated from these permissions, we do grant this set of permissions to T-5 containers, which target non-responsive/slow server issues, since these usually involve killing resource-consuming processes.

For T-6, covering software issues, we match a perforated container with ITFS-monitored access to the root filesystem of the target machine. The network view of such a container includes only the organizational software repository and monitored access to a whitelist of websites.

The perforated container for T-9, which handles SSH, VNC, and batch computing problems, shares the corresponding configuration files (located under /etc/ and in the home directory) with the host, and has a limited network view, which includes the target machine (for SSH and VNC) and the organizational batch computing server (for batch computing). To enable service restarts after configuration fixes, it is granted the process management permission set. Note that this container is deployed both on the user and the target machines, since configurations might need to be fixed in both of them.

T-7 is the container for VM cloud issues. The organizational VM cloud is managed using an EC2-style GUI that can create, reboot, terminate, and change resource allocations of the hypervisors. These operations cover most of the VM cloud-related tickets, without the IT personnel having to log into any VMs. However, when creating a new VM from a ready and signed image, the IT person must access it in order to configure its ownership properties. With WatchIT included in the signed initial filesystem image of each VM, the T-7 perforated container is only exposed to the relevant ownership configuration files in /etc/. Thus, if a user requires a new VM with some software installed, she should create two tickets – one for a new VM on her name, and one for software installation in that VM.

For issues that do not match any of the classes, we build a fully isolated container, T-11, thus tracking and logging all operations that are done while attending to the unclassified ticket.

7.1.3 Testing and Results. After extracting the ticket classes from historical data, during three months from December 2016 to February 2017, we collected and audited all 398 Linux-related tickets created in the system, excluding hardware failures and Windows/iOS related tickets. For each ticket, we recorded the operations that were performed while treating

ID	% of Total Tickets	Classif-ication Precision	% of Tickets Satisfied by P.Container	% of Tickets Used Permission Broker		
				Process Management	Filesystem	Network
T-1	9%	94%	94%	3%	-	3%
T-2	7%	95%	86%	-	-	14%
T-3	8%	94%	93%	-	-	7%
T-4	2%	100%	100%	-	-	-
T-5	5%	95%	89%	-	-	11%
T-6	30%	94%	91%	-	-	9%
T-7	10%	100%	97%	3%	-	-
T-8	3%	92%	75%	17%	-	17%
T-9	21%	98%	100%	-	-	-
T-10	3%	92%	100%	-	-	-
T-11	2%	80%	-			
Total	**100%**	**95%**	**92%**	**1%**	**-**	**7%**

Table 4: Results of attempting to use our custom-made perforated containers for handling tickets collected during a test period.

Container		Capabilities			
ID	Dist.	Process Management Permission Set	Home Directory	/etc/	Network Namespace
S-1	60%			X	
S-2	20%	X		X	
S-3	10%		X		
S-4	10%	X		X	X

(a) Custom-made perforated containers for Chef and Puppet scripts.

Container		Capabilities			
ID	Dist.	Process Management Permission Set	Statistic Tools	Filesystem Access	Network Namespace
S-5	80%		X		
S-6	20%	X			

(b) Custom-made perforated containers for cluster management scripts.

Figure 8: Perforated container tailoring for IT scripts.

it, and asked the IT team to classify it to one of the ten predefined categories. Thus, we created a database that includes for each ticket its free-text description, a classification, and the permissions required for handling it. Next, we check whether we can apply the operations performed for each ticket inside its corresponding perforated container. We also predict the class of each ticket using our LDA model, after applying spelling correction.

We present the results in Table 4. The first column shows the category distribution of the collected ticket set, as labelled by the IT department, and the second presents the prediction accuracy for tickets of each class. The third column shows, for each category, the percentage of tickets in this category that were completely satisfied by their corresponding custom-made perforated container. As the table shows, 92% of tickets could be handled from within the corresponding container. Considering the isolation of each container, we prevented full filesystem view in 62% of the cases, compartmentalized the process view in 36% of the tickets, and isolated the network view in 98% of the cases. Moreover, access to the world-wide web was made possible only in 32% of the tickets (T-6), and only to whitelisted websites. Furthermore, all filesystem and network accesses were monitored.

Tickets that did not match any of the predefined categories were classified as "Other" (T-11). These include rare IT requests such as partition resizing and driver updates. The latter constitutes only 0.5% of the tickets; they change our TCB and are indeed rare.

The next three columns in Table 4 details the causes for which the permission broker was employed. One example of such a ticket was classified as license problem since a user requested a license for a specific Matlab toolbox, but the toolbox was not installed on his machine. Since license-type perforated containers are isolated from the software repository, the IT needed the permission broker in order to install the toolbox.

7.2 IT Scripts

Chef and Puppet. We review twenty bash scripts used by the IT department. These scripts are intended for various purposes: time synchronization, permission and configuration verification, service restarts, etc. They are executed periodically and before ticket handling, using Chef and Puppet. They execute with root privileges and occasionally resolve the ticket without human intervention.

We examine the required isolation for each script and conclude that most of them access only specific configuration files, few of them engage with system processes, and a few require sharing the network namespace (for IP table operations). Overall, we group the scripts into four categories. Hence, as listed in Figure 8a, we build four different containers and map each of the scripts to a container that can run it under maximal isolation.

Cluster Management. We audit thirteen scripts used for automation and management of Apache Spark and IBM Swift clusters. These scripts mainly collect statistics, search for failures by reading system logs, and automate cluster operations like service restarts and system reboots. As presented in Figure 8b, we learn that a single, very limited perforated container can answer the needs of 80% of the scripts. This container should have access only to system logs and statistic tools (e.g., `mpstat`). The rest of the scripts handle system services and reboots, and are thus granted only the process management permission set. Note that these perforated containers are isolated from the network; as a result, tampered scripts can never leak information outside of the cluster.

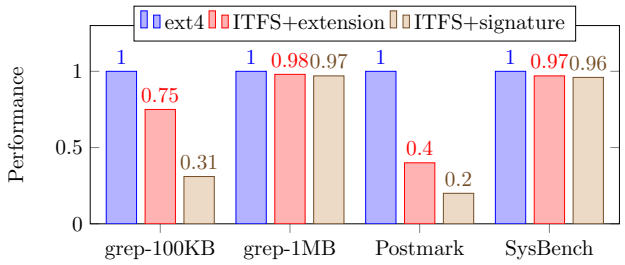


Figure 9: ITFS performance evaluation.

7.3 ITFS Evaluation

Our container software utilizes the Linux namespace mechanism and therefore enjoys the low overhead attributed to containers. That being said, as [36] shows, the use of ITFS, just like any other FUSE-based filesystem, may incur performance degradation.

We measure the ITFS overhead by performing a typical administration task, grep, on 25GB directories with average file sizes of (1) 1MB and (2) 100KB. We further use two other benchmarks, characterized by different workloads: (1) Postmark [22], configured to access many 5KB-256KB files; and (2) SysBench [6], which accesses a small number of large files. Our system runs Ubuntu 16.04 on Intel Core i7-4790 with 16GB RAM and SSD hard drive.

Figure 9 presents the results of executing these benchmarks on our system with three filesystem configurations: ext4 (baseline), ITFS with file-extension monitoring, and ITFS with file-signature monitoring. We see that the ITFS overhead depends on the sizes of the accessed files and on the monitoring rules. Overall, when engaging large files the performance is close to the baseline and under small file workload the ITFS overhead becomes more substantial. Note that ITFS mainly provides permission checking and does not intervene in the actual read or write operations. Therefore, if one wishes to improve its performance, one can employ a pass-through read/write approach as proposed in previous work [31].

8 RELATED WORK

The leading solutions for providing protection from insider threats are based on mandatory access control [1, 7], tainting processes that access classified information [23], and defining SDNs within the organization [5]. However, they all trust the system administrators; a rogue IT person can change their configuration and compromise the system.

SELinux [29] and others [12, 27] implement role-based access control models. Contrary to these, WatchIT adopts an ACL-like approach, which proved to be more practical. Furthermore, unlike WatchIT, they do not provide monitoring and network virtualization, and escalation is not possible in cases of insufficient permissions.

Security information management software such as [16, 18, 20, 42, 43, 46] helps organizations collect and analyze log and intelligence data in order to identify malicious activities. However, rather than *proactively* preventing attacks, they only perform after-the-fact analysis to *reactively* detect anomalies.

The Jail [24] and Zone [34] mechanisms are designed to enable multiple root users, each with a different view of the system. However, these are intended for isolating customers in server consolidation scenarios. Contrary to our approach, no actions can be performed on the host from within a Jail or a Zone, and they are not suited for administration.

Santos et al. [41] proposed an OS that suppresses superuser privileges and exposes a narrow management interface, thus protecting systems from untrusted administrators. The WatchIT approach, on the other hand, allows system administrators to retain their superuser privileges, thus causing minimal changes to IT workflow. Moreover, WatchIT does not require changes to the OS, and provides monitoring on all actions performed by IT personnel.

9 CONCLUSIONS

We proposed an approach for mitigating insider threats from the organizational IT department. Our strategy exploits containers' properties, but goes against their nature by perforating their isolation, and thus turns them into sandboxes for administration. We further implemented a proof-of-concept of our approach and provided a case study on a real IT department in which we custom-tailored perforated containers to the needs of the studied IT department. The isolation and permissions, as well as the ticket class granularity can be tuned for the needs of any organization that adopts our approach.

Finally, we note that WatchIT is not a panacea. Collusions can bypass WatchIT protection and group-targeted attacks remain efficient in the presence of WatchIT. That said, the mere fact that bypassing WatchIT requires collaboration with another insider or a sophisticated infection of a known website implies that WatchIT significantly raises the bar for attacks by an adversarial administrator.

10 ACKNOWLEDGMENTS

This research was funded in part by the Hasso-Plattner Institute (HPI), the Technion Funds for Security Research and the Technion Hiroshi Fujiwara Cyber Security Research Center. We heartily thank the IBM Research IT department for sharing their knowledge and database. We further thank our students Ron Blechner, Merav Natanson, Guy Barshatski, Bassam Yassin, and Hezi Banda for helping implement WatchIT components. Finally, we thank our shepherd, Ding Yuan, and the referees for their insightful reviews.

REFERENCES

[1] 2015. TOMOYO: A Security Module for System Analysis and Protection. (2015). http://tomoyo.osdn.jp/ Available at http://tomoyo.osdn.jp/.

[2] 2016. *Linux Programmer's Manual* (4.10 ed.).

[3] 2017. *Linux Containers*. Available at https://linuxcontainers.org/.

[4] 2017. *OpenVZ*. Available at openvz.org.

[5] Stefan Achleitner, Thomas La Porta, Patrick McDaniel, Shridatt Sugrim, Srikanth V. Krishnamurthy, and Ritu Chadha. 2016. Cyber Deception: Virtual Networks to Defend Insider Reconnaissance. In *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats (MIST '16)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/2995959.2995962

[6] Alexey Kopytov. 2016. SysBench - A Modular, Cross-Platform and Multi-Threaded Benchmark Tool. (2016). http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html

[7] Mick Bauer. 2006. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux J.* 2006, 148 (Aug. 2006), 13–. http://dl.acm.org/citation.cfm?id=1149826.1149839

[8] Eric Biederman and Karel Zak. 2017. *nsenter - Run Program With Namespaces of Other Processes*. Linux Man Pages.

[9] David M. Blei. 2012. Probabilistic Topic Models. *Commun. ACM* 55, 4 (April 2012), 77–84. https://doi.org/10.1145/2133806.2133826

[10] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (March 2003), 993–1022. http://dl.acm.org/citation.cfm?id=944919.944937

[11] Balázs Bucsay. 2015. Chw00t: Breaking Unices' chroot() Solutions. (2015). Available at https://github.com/earthquake/chw00t.

[12] M. Bugliesi, S. Calzavara, R. Focardi, and M. Squarcina. 2012. Gran: Model Checking Grsecurity RBAC Policies. In *2012 IEEE 25th Computer Security Foundations Symposium*. 126–138. https://doi.org/10.1109/CSF.2012.29

[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (July 2009), 58 pages. https://doi.org/10.1145/1541880.1541882

[14] Chef 2017. *Chef – Automate Your Infrastructure*. Chef. Available at www.chef.io.

[15] You Chen and Bradley Malin. 2011. Detection of Anomalous Insiders in Collaborative Environments via Relational Analysis of Access Logs. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy (CODASPY '11)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1943513.1943524

[16] Cisco. 2017. Cisco™Security MARS. (2017).

[17] Sharon Gaudin. 2006. Ex-UBS Systems Admin Sentenced To 97 Months In Jail. (December 2006). www.informationweek.com/ex-ubs-systems-admin-sentenced-to-97-months-in-jail/d/d-id/1049873?

[18] Gaurang Gavai, Kumar Sricharan, Dave Gunning, Rob Rolleston, John Hanley, and Mudita Singhal. 2015. Detecting Insider Threat from Enterprise Social and Online Activity Data. In *Proceedings of the 7th ACM CCS International Workshop on Managing Insider Security Threats (MIST '15)*. ACM, New York, NY, USA, 13–20. https://doi.org/10.1145/2808783.2808784

[19] Google. 2017. gRPC: A High Performance, Open-Source Universal RPC Framework. (2017). Available at https://grpc.io/.

[20] Hewlett Packard. 2017. ArcSight ESM. (2017).

[21] IBM® X-Force Research 2016. *2016 Cyber Security Intelligence Index*. IBM® X-Force Research.

[22] Jeffrey Katcher. 1997. *Postmark: a New File System Benchmark*. Technical Report. TR3022, Network Appliance.

[23] Ryan V. Johnson, Jessie Lass, and W. Michael Petullo. 2016. Studying Naive Users and the Insider Threat with SimpleFlow. In *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats (MIST '16)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/2995959.2995960

[24] Poul-Henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the Omnipotent Root. In *In Proc. 2nd Intl. SANE Conference*.

[25] Jesse D. Kornblum. 2009. Implementing BitLocker Drive Encryption for Forensic Analysis. *Digit. Investig.* 5, 3-4 (March 2009), 75–84. https://doi.org/10.1016/j.diin.2009.01.001

[26] David Kravets. 2008. San Francisco Admin Charged With Hijacking City's Network. (July 2008). www.wired.com/2008/07/sf-city-charged/

[27] Ninghui Li, John C. Mitchell, and William H. Winsborough. 2002. Design of a Role-Based Trust-Management Framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP '02)*. IEEE Computer Society, Washington, DC, USA, 114–130. http://dl.acm.org/citation.cfm?id=829514.830539

[28] Wolfgang Mauerer. 2008. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK.

[29] Bill McCarty. 2004. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc.

[30] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[31] Reddy Nikhilesh. 2016. FUSE: Add Support for Passthrough Read-/Write. (February 2016). http://fuse.sourceforge.net/ Available at https://lwn.net/Articles/674286/.

[32] Angela Orebaugh, Gilbert Ramirez, Jay Beale, and Joshua Wright. 2007. *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Syngress Publishing.

[33] Lennart Poettering, Kay Sievers, Harald Hoyer, Daniel Mack, Tom Gundersen, and David Herrmann. 2016. systemd-nspawn. (November 2016). Available at wiki.archlinux.org/index.php/Systemd-nspawn.

[34] Daniel Price and Andrew Tucker. 2004. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th USENIX Conference on System Administration (LISA '04)*. USENIX Association, Berkeley, CA, USA, 241–254. http://dl.acm.org/citation.cfm?id=1052676.1052707

[35] Puppet 2017. *Puppet - The shortest path to better software*. Puppet. Available at https://puppet.com/.

[36] Aditya Rajgarhia and Ashish Gehani. 2010. Performance and Extension of User Space File Systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*. ACM, New York, NY, USA, 206–213. https://doi.org/10.1145/1774088.1774130

[37] Nikolaus Rath. 2017. FUSE: Filesystem in Userspace. (2017). http://fuse.sourceforge.net/ Available at http://fuse.sourceforge.net/.

[38] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA '99)*. USENIX Association, Berkeley, CA, USA, 229–238. http://dl.acm.org/citation.cfm?id=1039834.1039864

[39] Malek Ben Salem, Shlomo Hershkop, and Salvatore J. Stolfo. 2008. *A Survey of Insider Attack Detection Research*. Springer US, Boston, MA, 69–90. https://doi.org/10.1007/978-0-387-77322-3_5

[40] Jerome H. Saltzer. 1974. Protection and the Control of Information Sharing in Multics. *Commun. ACM* 17, 7 (July 1974), 388–402. https://doi.org/10.1145/361011.361067

[41] Nuno Santos, Rodrigo Rodrigues, and Bryan Ford. 2012. Enhancing the OS Against Security Threats in System Administration. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*. Springer-Verlag New York, Inc., New York, NY, USA, 415–435. http://dl.acm.org/citation.cfm?id=2442626.2442653

[42] Splunk. 2017. Splunk™User Behavior Analytics. (2017).

[43] Symantec. 2013. Symantec™Security Information Manager. (2013).

[44] Bob Toxen. 2002. *Real World Linux Security* (2nd ed.). Prentice Hall Professional Technical Reference.
[45] Bob Van Zant. 2017. SSH Certificate Authority. (2017). Available at github.com/cloudtools/ssh-ca.
[46] Varonis. 2017. "Varonis™Enterprise Security". (2017).
[47] Joseph Verble. 2014. The NSA and Edward Snowden: Surveillance in the 21st Century. *SIGCAS Comput. Soc.* 44, 3 (Oct. 2014), 14–20. https://doi.org/10.1145/2684097.2684101
[48] W. T. Young, A. Memory, H. G. Goldberg, and T. E. Senator. 2014. Detecting Unknown Insider Threat Scenarios. In *Security and Privacy Workshops (SPW), 2014 IEEE*. 277–288. https://doi.org/10.1109/SPW.2014.42