Poster: TCLP: Enforcing Least Privileges to Prevent Containers from Kernel Vulnerabilities

Suyeol Lee*
95leesu@kaist.ac.kr
KAIST

Junsik Seo*
js0780@kaist.ac.kr
KAIST

Jaehyun Nam
namjh@kaist.ac.kr
KAIST

Seungwon Shin
claude@kaist.ac.kr
KAIST

ABSTRACT

While containerization has emerged as a lightweight approach to package, deploy, and run legacy applications in a resource-efficient manner, the shared kernel-resource model used by containers introduces critical security concerns. Specifically, the abuse of system calls by a compromised container can trigger the security vulnerabilities of a host kernel. Unfortunately, even though existing solutions provide powerful protection mechanisms against such issues, how to define the capabilities of containers is still up to operators. In this work, we thus introduce TCLP, a dynamic analysis system that helps operators configure the least capabilities of containers to protect not only themselves but also a host. TCLP monitors the system calls triggered by containers in run time and finds the least capabilities required to run the containers based on the collected system calls. Finally, operators configure the minimal capabilities discovered by TCLP for their containers, reducing the risk of kernel vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

Virtualization Security; Containerization; Container Privilege

ACM Reference Format:
Suyeol Lee, Junsik Seo, Jaehyun Nam, and Seungwon Shin. 2019. Poster: TCLP: Enforcing Least Privileges to Prevent Containers from Kernel Vulnerabilities. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3319535.3363282

1 INTRODUCTION

These days, containerization has emerged as one of the most important technologies for cloud environments because of its high resource efficiency compared to hypervisor-level virtualization. It provides a lightweight operating-system-level virtualization that only provides the bare minimum that containerized applications require to run as intended while all containers share the same kernel of a host with other containers. Thus, containers could package, deploy, and run legacy applications in an efficient and flexible way.

Despite such benefits, containerized applications and even a host can fall prey to security vulnerabilities due to the shared kernel-resource model used by containers. This shared model enables compromised container to exploit the host kernel by employing system calls to trigger some kernel vulnerabilities, producing destructive impacts on not only a host but also other containers. For example, a container can craft a system call with specially designed arguments to make unexpected behaviors such as memory leak (CVE-2016-0728 [3]). It can also abuse a race condition to write arbitrary contents into the shared files with the host by triggering two system calls simultaneously even though these files are read-only (CVE-2016-5195 [6]). Furthermore, a container can modify its capabilities to gain elevated privileges by exploiting a certain system call (CVE-2017-5123 [4, 9]). Likewise, while the shared kernel-resource model brings lots of efficiency gains, it can introduce critical security concerns.

To address those security concerns, several security solutions such as AppArmor[2], SELinux[7] have been released to limit the capabilities of containers in a more fine-grained way. Unfortunately, although they provide powerful security mechanisms, they do not give any insight into how to minimize capabilities to prevent containers and a host from kernel vulnerabilities. Thus, operators should carefully consider all possible security concerns and configure the capabilities of all containers by themselves.

In this work, we introduce TCLP, a dynamic analysis system that provides the capability guidelines (i.e., the minimum capabilities) of the given containers by operators. To extract the least capabilities required to run containerized applications, TCLP first builds a mapping table of system calls and their corresponding capabilities. Then, while launching containers, it traces all system calls triggered by the containers. Finally, it generates the capability list required to run those containers by searching capabilities mapped to collected system calls. In the end, operators only need to configure the least capabilities discovered by TCLP for their containers.

2 BACKGROUND AND MOTIVATION

2.1 Linux Capabilities

Capabilities are one of the key security mechanisms in Linux kernel, which provide fine-grained control over root permissions. For example, if the CAP_CHOWN capability of a container is disabled,

*The first two authors contributed equally to this work.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11.
https://doi.org/10.1145/3319535.3363282

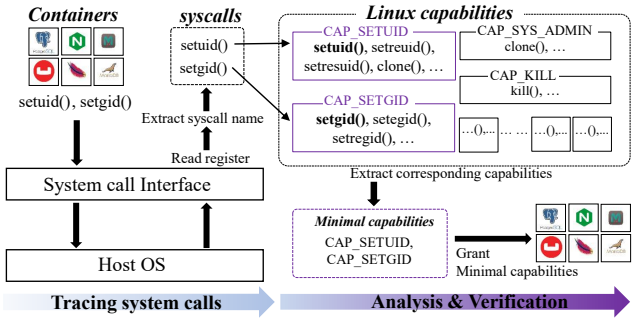

Figure 1: Overall Workflow of TCLP

the container can no longer change the UIDs and GIDs of internal files. Since Linux kernel 3.0, *38* capabilities have been implemented inside of a kernel and have controlled more than 110 system calls. For the security and flexibility of containerized applications, containers use those capabilities to restrict the use of sensitive system calls, and *15* out of *32* capabilities are set in containers by default [1] while these capabilities still allow lots of critical system calls to containers.

2.2 Motivating Examples

Although containers only have the limited number of capabilities by default, an infected container by an attacker can still abuse system calls allowed by default capabilities to achieve certain goals (e.g., privilege escalation and arbitrary code execution). Here, we present two examples that abuse capabilities to exploit the host kernel.

Privilege escalation using memory leakage: While containers generally run as non-privileged users, malicious containers can abuse vulnerable system calls to escalate their privileges. For example, they can exploit the `keyctl` system call, which is used for the key management in the Linux kernel, to cause memory leakage and gain privileges (CVE-2016-0728 [3]). In this example, only CAP_SETUID and CAP_SYS_ADMIN capabilities are required.

Arbitrary code execution using race condition: Some other containers can abuse certain system calls to execute arbitrary codes as well. For instance, they can keep triggering two system calls (e.g., `madvise` and `ptrace`) to incur a race condition on a certain memory (e.g., executable memory region) (CVE-2016-5195 [6]). Then, the containers can write arbitrary codes in that memory no matter whether the memory is read-only or not. Here, those containers only need to have two capabilities: CAP_SYS_ADMIN and CAP_SYS_PTRACE.

3 DESIGN

3.1 Overview

From the motivating examples discussed in Section 2.2, we see that a malicious container can cause critical security problems using system calls and already existent kernel vulnerabilities even through system calls allowed by given capabilities are triggered with valid arguments in a proper way. Hence, it is important to discover and disable all capabilities not used by containerized applications in advance. In this work, we thus introduce TCLP, a dynamic analysis system that extracts the least capabilities required to run given containerized applications.

Figure 1 shows the overall workflow of TCLP. To figure out the capabilities required to run containerized applications, TCLP goes through three major steps: tracing system calls, finding the capabilities for triggered system calls, and verifying the functionalities of given containers with discovered capabilities. Here, we briefly explain how each step works.

3.2 Mapping system calls on capabilities

Before extracting the capabilities required for given containers, we need to understand what kinds of system calls each capability controls. Unfortunately, there are no explicit mappings between capabilities and system calls, and multiple capabilities can control the same system calls as well. Thus, TCLP first monitors which capabilities are checked when system calls are triggered in run time. Then, it builds its own capability-and-system-call mapping table based on the pairs of monitored capabilities and system calls.

3.3 Tracing system calls from each container

In order to trace system calls triggered by containers not host processes, it is important to figure out which processes are actually executed inside of given containers since there is no difference between them in the view of a host. Thus, TCLP first keeps monitoring the process lists inside of given containers using the process hierarchy structure where the top parent processes are `containerd-shim`, and it traces all system calls triggered by those processes using `strace` [8].

3.4 Analyzing the use of system calls

After tracing all system calls triggered by each container, TCLP first extracts a unique set of system calls. Then, it finds the capabilities corresponding to each system call from the capability-and-system-call mapping table. At this point, if a system call belongs to multiple capabilities, TCLP chooses the capability that has the smallest number of system calls to minimize the possibility of abusing the other system calls that the capability allows.

3.5 Verifying the discovered capabilities

As the last step, TCLP launches given containers with a set of discovered capabilities, and it monitors whether any failed system calls are detected or not. If there is no failed system call, TCLP informs operators of the discovered capabilities since they are the least capabilities to run containerized applications. Otherwise, TCLP adds the capabilities for newly discovered system calls into the capability set, and it repeats the verification step again.

4 PRELIMINARY RESULTS

To show the effectiveness of TCLP, we use 12 container images retrieved from Docker Hub, which is the most popular container image repository[5]. Here, we discuss the trends of the required capabilities for those containers, which are discovered by TCLP.

The uses of Linux capabilities and system calls: Figure 2 presents the number of required capabilities and collected system calls from those containers. From the results, we see that containers generally trigger 41.6 system calls on average for their operations.

Table 1: Linux capabilities generally required by containers

Set type	Linux Capabilities	System Calls	Description
Default	CAP_SETGID	setgid()	Make arbitrary manipulations of process GIDs
	CAP_SETUID	setuid()	Make arbitrary manipulations of process UIDs
File access	CAP_FOWNER	open(), openat()	Bypass permission checks on operations
Networking	CAP_NET_BIND_SERVICE	bind()	Bind a socket to internet domain privileged ports (e.g., well-known ports)
	CAP_NET_RAW	socket()	Open a non-TCP/UDP socket

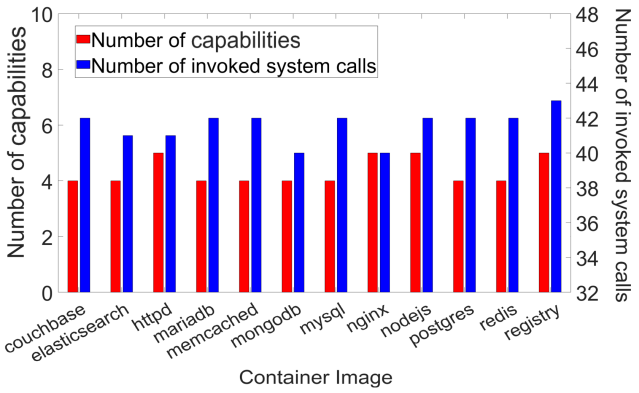

Figure 2: The uses of Linux capabilities and system calls

Here, we note that most of system calls (e.g., `read`, `write`, `lseek`, `connect`, and `execve`) do not require any capabilities, which means that they need no privileges. Besides them, certain system calls (e.g., `socket`, `setsockopt`, `open`, `mmap`, and `mmunmap`) require capabilities, and they mostly need 4.3 capabilities (e.g., `CAP_NET_ADMIN`, `CAP_SYS_ADMIN`, `CAP_FOWNER`, and `CAP_IPC_LOCK`) on average while containers basically have 15 capabilities.

Capabilities generally required for applications: Here, we classify discovered capabilities into three privilege sets according to their purposes (i.e., default, file access, and networking), and see why they are required to run containerized applications. Table 1 shows these capabilities. The first set of capabilities is used to create separated spaces (i.e., namespaces) for containers from a host. Especially, containers utilize these capabilities to change the UIDs and GIDs inside of containers.

The second set of capabilities is generally used to access files in a host. There are some cases that containers need to access the files located in a host. For example, all of the 12 containers we used require certain configuration files, and we place these files in the file system of the host. In this case, the `CAP_FOWNER` capability is required to import the files inside of containers.

The last set of capabilities is about networking. Many of containers provide Web services and may use well-known ports for the services. However, the Linux kernel basically blocks binding to a port less than 1024 without privileges. Thus, if containers need to bind to a port less than 1024, they require the CAP_NET_BIND_SERVICE capability. Also, there are some cases that containers need to handle non-TCP/UDP packets (e.g., ICMP packets). In this case, containers require the CAP_NET_RAW capability to create raw sockets.

5 CONCLUSION AND FUTURE WORK

While containers have adopted the shared kernel-resource model to achieve the maximum resource efficiency, it also allows containers to abuse system calls to trigger kernel vulnerabilities and produce destructive impacts on both other containers and the host. To address such security problems, we have proposed TCLP that traces all system calls executed by containers and extracts the minimum capabilities that only allow the system calls required to run containerized applications. Finally, operators can configure their containers based on the least capabilities discovered by TCLP.

From our preliminary results, we see that many of containers actually do not need a large number of capabilities to run their applications. More specifically, the results show that default capabilities still give excessive privileges to containers, and TCLP can help operators to properly configure the capabilities of their containers.

In terms of our future work, since we currently rely on the capabilities and system calls we collected, there could be some missing system calls that containers actually use even though TCLP may be able to supplement this issue by repeating its verification process. Hence, the static analysis of container images would be required along with our dynamic analysis. Also, we see that there are lots of system calls that do not require high privileges. However, it is possible that those system calls are also abused to trigger some other kernel vulnerabilities. Thus, the restrictions on non-privileged system calls (e.g., Seccomp policy generation) would be needed to further reduce the risk of kernel vulnerabilities.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP), a grant funded by Korea government (Ministry of Science and ICT) (no. 2016-0-00078, Cloud Based Security Intelligence Technology Development for the Customized Security Service Provisioning).

REFERENCES

[1] 2019. Docker Container Specification. https://github.com/docker/libcontainer/blob/master/SPEC.md.
[2] AppArmor. 2019. https://gitlab.com/apparmor.
[3] CVE-2016-0728. 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0728.
[4] CVE-2017-5123. 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5123.
[5] Docker. 2019. Docker Hub. https://hub.docker.com.
[6] Perception Point. 2016. https://perception-point.io/resources/research/analysis-and-exploitation-of-a-linux-kernel-vulnerability.
[7] SELinux Project. 2019. http://selinuxproject.org/page/Main.
[8] Linux syscall tracer. 2019. https://strace.io.
[9] TwistLock. 2017. Escaping Docker container using waitid. https://www.twistlock.com/2017/12/27/escaping-docker-container-using-waitid-cve-2017-5123.