Enhancing Container Security by Logging the Unpopular Paths in the Operating System Kernel

Yiwen Li
New York University
liyiwen@nyu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

Justin Cappos
New York University
jcappos@nyu.edu

Abstract

Containers, such as Docker and LinuxKit, are widely used because of the perceived isolation and security they provide against the potentially malicious or buggy user programs running inside of them. However, this perception may be false. It is possible to trigger zero-day kernel bugs from inside of a container, which could lead to such security problems as privilege escalation. The reason for this vulnerability is that these containers are allowed access to rarely executed paths in the host operating system kernel, which often contain bugs. A previous study [5] has proven the value of limiting kernel access to only the frequently used "popular paths," as they contain fewer security bugs. Therefore, if an application could run using only these paths, the security of the container would be greatly enhanced.

In this paper, we show that the popular paths metric can be used to improve the security of existing containers, such as LinuxKit. We designed and implemented a ***Secure Logging System***, which first finds the popular paths for the LinuxKit kernel by profiling popular Docker containers from Docker Hub [3], and then creating a ***secure logging kernel*** designed to run only on those secure paths. Whenever a line of code from an unpopular path is to be executed, this kernel generates warning messages to alert users about potential security breaches. Our evaluation verified that the containers tested were able to run their default workload normally using our secure logging kernel, touching the popular paths 99% of the time. As an examination of 50 Linux kernel CVE security vulnerabilities confirmed only 6% of them were present in the popular paths, this means our design is a more inherently secure alternative to conventional containers. Furthermore, as our performance evaluation shows, running Docker containers using our secure logging kernel only incurs about 2% runtime overhead, and around 0.5% extra cost of memory space. Therefore, the new design can provide better security without any significant loss of efficiency.

1 Introduction

Containers continue to grow in popularity, with one industry survey suggesting the technology could become a $2.7 billion market by 2020 [1]. This wide-spread adoption of containers, such as Docker [2] and LinuxKit [6] can be attributed to the perceived isolation and security they provide to the user programs running inside of them. However, this perception may be false, as it is possible to trigger zero-day kernel bugs from inside of a container [cite]. And since the kernel is shared by containers and the host system, exploitation of such bugs could lead to severe security problems, such as privilege escalation [cite a few examples of exploitation].

The problem lies in allowing containers access to the underlying host operating system kernel where zero-day bugs could be encountered [cite]. To this point, there has been little or no efforts to limit such contact, in part because there was no efficient way to identify where the bugs might be. If one knew that part of the kernel did not contain zero-day bugs— or at the very least was less likely to host this threat— then restricting a container to touch only this part of the kernel would greatly improve security.

A previous study [5], has shown that there is a powerful correlation between the popularity of a kernel code path and its likelihood to contain a security flaw. Furthermore, the study [5] demonstrated that these frequently used kernel paths can be leveraged to design and build secure virtualization systems. Building on these results, we ask the question, can we apply the popular paths metric to securing existing containers? And, if such an application is possible, could it provide stronger security and truer isolation than the existing design of container systems permits?

In this paper, we document our efforts to answer those questions by studying the feasibility of using the popular paths metric to secure the LinuxKit container toolkit. This entailed finding a way to identify the parts of the underlying kernel that are less likely to contain security bugs, demonstrating that existing containers could operate exclusively on these paths, and developing a method to warn users away from code found in the less-used paths.

We were able to demonstrate a systematic way to gather data on popular paths, and affirmed this data is representative of hundreds of popular Docker containers from Docker Hub [3]. Next, we implemented a secure logging kernel designed to generate warning messages whenever a line of code from the unpopular paths is to be executed. By doing so, users can detect potentially dangerous program behaviors and make decisions about whether any attempt to execute unpopular paths should be blocked.

An evaluation of our findings affirmed that popular Docker containers can run normally using our secure logging kernel. For the official Docker containers' default workload, the popular paths were used 99% of the time. And whenever unpopular paths were reached, our secure logging kernel was able to log these paths and warn users about the potentially dangerous program behaviors. Our security evaluation was able to confirm that the using the popular paths to run Docker containers can effectively prevent most kernel vulnerabilities from being triggered, as only three of the 50 CVE kernel security vulnerabilities we checked for were found in the popular paths of LinuxKit. And, in our performance evaluation, we found that running our secure logging kernel only incurred about 2% of runtime overhead. The memory space overhead of our secure logging kernel was about 0.5% (193.5MB-192.4MB/192.4MB = 0.57%). Thus, greater security could be achieved with very little perceived difference in operation efficiency or cost.

2 Background and Motivation

The concept of OS virtualization, where an operating system runs programs in an isolated space over a kernel, has been around for several decades [8]. Yet, the emergence of container programs [11], which run off the kernel's OS and therefore have a lower overhead, has spurred growth in the popularity of these machines, as Docker [2], LinuxKit [6], and Kubernetes [4], have seen wider adoption over the past few years. For example, Docker Hub [3] holds more than two million container images , and the most popular ones have been downloaded by more than ten million users. Many users prefer using containers over traditional full-scale virtual machines such as VMware workstation [10] and VirtualBox [9], because containers allow them to test and develop their software programs with relatively low overhead. In addition, the idea of running their programs in a "contained" environment tends to make developers feel they have secured them against all threats.

Existing container systems use several mechanisms to provide isolation and security to the programs running inside of them. Namespaces [8] were a modern Linux kernel feature, introduced between Linux kernel version 2.6.15 and 2.6.26, to provide a global system resource abstraction that helps achieve isolation between different containers. Docker and LXC [9] containers also used namespaces to provide isolation. Control groups [10] are another key Linux feature that enables accounting and limiting important system resources, such as memory, CPU, and disk I/O, to each container. In addition, this feature helps prevent denial-of-service attacks. Linux kernel capabilities [11] are used by a number of container systems, including Docker. Capabilities allow containers restrict permissions on specific resources, such as network and file system. Modern Linux kernels also have other kernel hardening systems, such as AppArmor [12], SELinux [13], GRSEC [14], PAX [15], etc., that may be leveraged by containers to provide some extra safety. GRSEC and PAX add safety checks at compile-time and run-time of the kernel. Docker ships a template that works with AppArmor, and also has SELinux policies working with Red Hat [16]. These kernel hardening systems are mostly access control mechanism that provides safety in the similar way that capabilities do.

While containers leverage existing mechanisms to try to improve security, there is yet one fundamental issue that is really challenging to address. The problem is, as acknowledged above, that unlike ia full-scale virtual machine, the operating system kernel is shared among all containers and the host. This magnifies the damage a kernel vulnerability could cause. If a bug in the kernel was triggered from one guest container, it could result in a security problem for other containers, and even the host, breaking all the isolation that users expect to get from a container.

The threat of zero-day kernel vulnerabilities is the main motivation behind this work. This research initiative was launched to prevent zero-day kernel bugs from being triggered in containers, while still allowing containers to perform their desired workload. Having already established in a previous paper that implementing the popular paths metric i could identify and neutralize the threat of these bugs [3] we theorized that container security can be enhanced in the same manner. To prove this hypothesis, we created a new design implementation for container security that is described in detail in the following section.

3 Design and Implementation

3.1 Design of Our "Secure Logging System"

Designing an implementation of the popular paths metric for use with containers required two separate tasks. First, we had to identify the popular paths for the underlying host operating system. And, second, once we knew where they were, second, we needed a way to log / block the non-popular paths to keep containers aways from these less-used and potentially buggy code paths. To address these issues, we came up with a dual module approach we call the Secure Logging System (Shown in Diagram 1).

Our system has two main modules, one is the "Kernel Profiler", the other one is the "Kernel Modifier". Our system operates in the following way: first, the Kernel Profiler will obtain the popular paths kernel trace for the given container system. Here, we define the "popular paths kernel trace" as the lines of code in the host kernel that are executed when running the default / regular workload (such workload is often defined in the configuration files that come with the container images) of a set of the most popular containers (ranked by the number of user downloads.) The key insight of our design is that this popular paths kernel trace we got should contain fewer kernel vulnerabilities, based on the findings in the Lock-in-Pop paper [3]. Second, our Kernel modifier leverages this popular paths data to enhance security of the kernel, by instrumenting the rarely used non-popular paths to either generate security warning messages, or block code execution on these potentially dangerous code

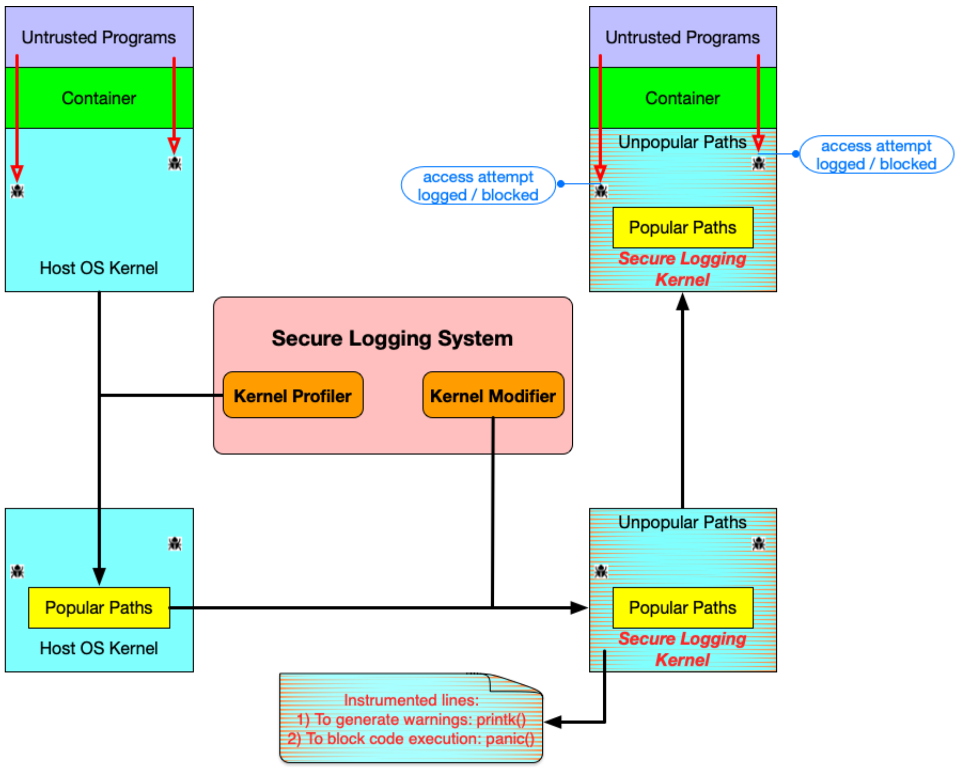

Figure 1. Design Overview of Our Secure Logging System

paths. Finally, our Secure Logging System produces a Secure Logging Kernel, which is an instrumented Linux kernel with security checks inserted at the non-popular paths. This Secure Logging Kernel can then be used to run containers without any changes to the containers themselves, and help prevent kernel vulnerabilities from being triggered.

3.2 Implementation

Our Secure Logging System has two main modules, the Kernel Profiler and the Kernel modifier. We now describe how each of them was implemented.

3.2.1 Implementation of the Kernel Profiler

The Kernel Profiler is designed to collect the kernel trace of running user containers. Especially, we are interested in obtaining the popular paths kernel trace. In this work, the popular paths kernel trace refers to the lines of code in the underlying host operating system kernel that were executed when running the regular workload (defined in the configuration files of the corresponding container images) of popular (determined by the number of user downloads) containers. In our implementation, we chose to run our Kernel Profiler on popular Docker containers from Docker Hub [4]. Docker Hub provides millions of Docker container images, and the popular ones are downloaded more than ten millions times. Our Kernel Profiler works in the following way:

1. The Linux kernel used to run the Docker containers will first be recompiled with the Gcov [17] kernel profiling feature enabled.
2. Our Kernel Profiler used the LinuxKit toolkit to run the Docker containers on the Gcov-enabled Linux kernel. To run the Docker containers, the Kernel Profiler first generate the configuration file for LinuxKit, in which the task container will be defined, along with a data container responsible for collecting, storing, and transferring the kernel trace data.
3. When booting the LinuxKit virtual machine, the Kernel Profiler has a script to start running the workload of the task container. And then, the Kernel Profiler will collect the kernel trace data (gcda files generated by Gcov), stored at "/sys/kernel/debug/gcov", and transfer the data to the host system for further use.
4. From the host system, our Kernel Profiler used the lcov [18] tool to process the kernel trace data we collected, and will generate the formatted data about which lines of code were executed in the kernel source files, to be used by our Kernel Modifier.

3.2.2 Implementation of the Kernel Modifier

The Kernel Modifier is responsible for modifying the Linux kernel to insert security logging code at the non-popular paths. It needs to first identify the correct places in the Linux

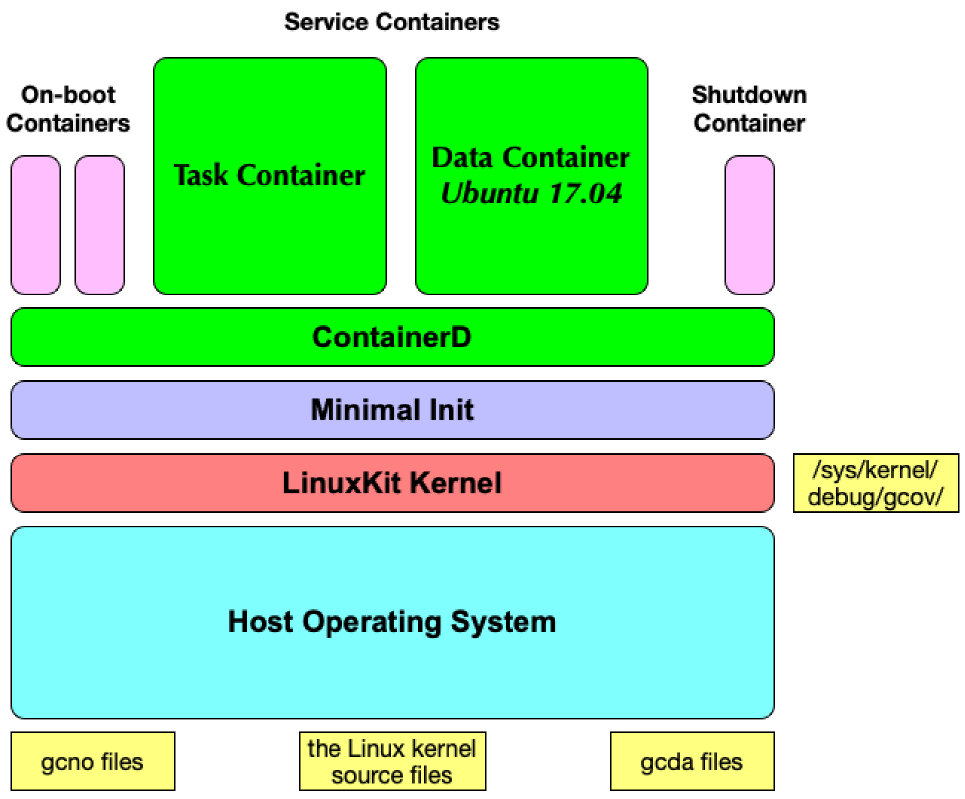

Figure 2. Implementation of the Kernel Profiler

kernel source tree to be instrumented, and then perform the code modification. The Kernel Modifier has three main parts, the Clang compiler, the Clang analyzer, and the kernel instrumenting tool (shown in Diagram 3). It operates in the following way:

1. First, we used the Clang C compiler from the Clang / LLVM project [19] along with BEAR [20] tool to compile the Linux kernel source. BEAR can help us generate a compilation database, which contains all the compile flags and options used during the kernel compilation process. And this compilation database will be used by our Clang analyzer to perform source code parsing in our next step.
2. Once we got the compilation database for the Linux kernel, next, we used our Clang analyzer to perform static analysis on the kernel source code to obtain the control flow graph for each function in the source code files. Our Clang analyzer leveraged Clang's LibTooling [21] to construct the AST tree from the kernel source code, and then obtain the control flow graph. From the control flow graph, we can identify the corresponding source line number for each basic block. And the beginning lines of all the basic blocks in the source files are our candidates for adding security logging code at this point.
3. Next, our kernel instrumenting tool will perform the actual kernel modification based on the basic blocks we obtained, and the popular paths data we collected. The algorithm for our kernel modification is that we go through the basic blocks in the kernel source files. If none of the lines in a basic block was in the popular paths data, we consider this basic block as a non-popular basic block, and our kernel instrumenting tool will add the security logging code at the beginning of this non-popular basic block. One optimization strategy we conducted here is that if we discover that an entire function has none of its lines in the popular paths data, we consider this function as a non-popular function, and then we just add our security logging code once at the beginning of this function, avoiding adding redundant and unnecessary code inside this function.
4. Finally, our Kernel Modifier will produce the Secure Logging Kernel. And It can be directly used to run Docker containers in the LinuxKit VM.

4 Evaluation

In this evaluation, our goal is to demonstrate that it is feasible to apply the popular paths metric onto existing container

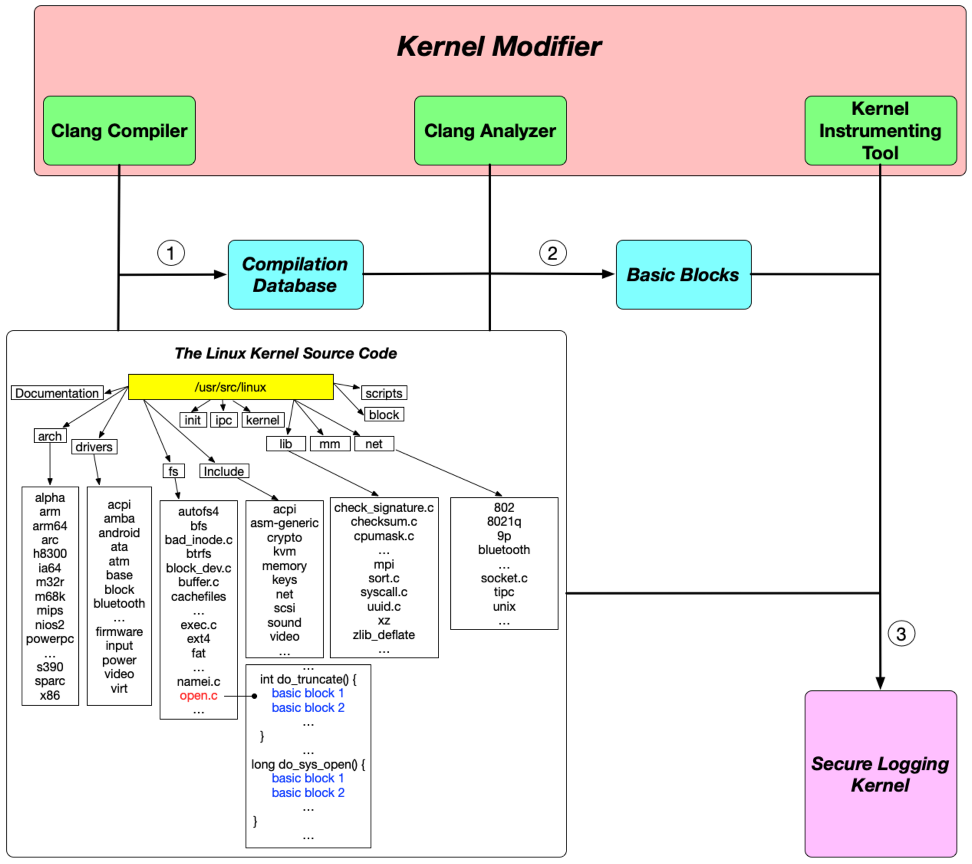
Kernel Modifier
Clang Compiler
Clang Analyzer
Kernel Instrumenting Tool
1
Compilation Database
2
Basic Blocks
The Linux Kernel Source Code
/usr/src/linux
Documentation
scripts
block
init
ipc
kernel
arch
drivers
lib
mm
net
fs
Include
alpha arm arm64 arc h8300 ia64 m32r m68k mips nios2 powerpc ... s390 sparc x86
acpi amba android ata atm base block bluetooth ... firmware input power video virt
autofs4 bfs bad_inode.c btrfs block_dev.c buffer.c cachefiles ... exec.c ext4 fat ... namei.c open.c ...
acpi asm-generic crypto kvm memory keys net scsi sound video ...
check_signature.c checksum.c cpumask.c ... mpi sort.c syscall.c uuid.c xz zlib_deflate ...
802 8021q 9p bluetooth ... socket.c tipc unix ...
... int do_truncate() { basic block 1 basic block 2 ... } ... long do_sys_open() { basic block 1 basic block 2 ... } ...
3
Secure Logging Kernel

Figure 3. Implementation of the Kernel Modifier

designs, such as Docker and LinuxKit. Furthermore, by applying this metric, we show that containers will have stronger security and isolation, while still maintaining functionality and performance.

To demonstrate this, we set out to answer the following questions:

- Can real-world containers run only on the popular paths? (Section 4.1)
- Does restricting access only to the popular paths improve the security of running containers? (Section 4.2)
- What is the performance overhead of only using the popular paths? (Section 4.3)

4.1 Usability Evaluation

We conducted experiments to test the feasibility of running real-world containers only using the popular paths. The goal here is to verify if users can still run their containers with their existing configuration and commands, using our secure logging kernel instead of the original Linux kernel. The secure logging kernel was created by our Secure Logging System.

Experimental Setup. First, we collected the kernel trace data by running the 100 most popular (ranked by the number of user downloads) Docker containers from Docker Hub. We use this dataset as our popular paths training set. Next, our Secure Logging System used our training set data to instrument the Linux kernel, and generated our secure logging kernel. Finally, we ran another 100 popular Docker containers, our testing set, on our security logging kernel.

In our experiment, we ran the popular Docker containers inside of a LinuxKit version 0.2+ machine, which was built using Docker version 18.03.0-ce. Our host operating system was Ubuntu 16.04 LTS, with Linux kernel 4.13.0-36-generic. A QEMU emulator version 2.5.0 served as the local hypervisor.

Results. This is our results.

4.2 Security Evaluation

The goal of our security evaluation is to answer the question: does restricting access only to the popular paths improve the security of running containers? We already es We try to answer this question by studying how many CVE kernel vulnerabilities were present in the popular paths of the LinuxKit VM.

Experimental Setup. In this paper, we used the Linux kernel version 4.14.24 when running the LinuxKit VM. We examined a list of 50 CVE kernel vulnerabilities (Table 1). Our methodology for getting this list is to obtain the CVE vulnerabilities for Linux kernel version 4.14.24 and version 4.14.x from the National Vulnerability Database [7]. These 50 CVE vulnerabilities were all the available ones for our targeted kernel versions at the time of our study (April 2019). For each of these CVE vulnerabilities, we looked at the patch that fixed the bug to identify the lines of kernel source code that could trigger this vulnerability. Next, we compared these lines against the popular paths kernel trace of LinuxKit, and identified which vulnerabilities were present in the popular paths.

Results. In our study, we found out that only three out of all the fifty CVE vulnerabilities were detected in the popular paths of LinuxKit (Table 1). And these three vulnerabilities detected were among commonly used kernel code that were hard to avoid by any program, and more likely to be patched since they were essential code that was used a lot. We describe these three kernel bugs found in the LinuxKit's popular paths in more details below.

4.3 Performance Evaluation

We evaluated both the run-time performance overhead and the memory space overhead of using our secure logging kernel.

Experimental Setup.

Results.

5 Limitation and Future Work

This is our limitation and future work.

What we should talk about here:

- the popular paths data collection: more containers and more tasks (especially)

6 Related Work

In this paper, we design and implement a Secure Logging System to reduce exposure of the rarely-used risky code in the kernel in order to enhance the security of containers running on top of the host kernel. In developing our secure logging strategy, we consulted previous studies in three areas: 1) metrics for vulnerability prediction, 2) techniques for enhancing kernel security, and 3) approaches for enhancing container security.

Estimating and predicting vulnerabilities.

Metrics that can identify the code that is most likely to contain vulnerabilities, help developers and researchers prioritize their efforts to fix bugs and improve security. Chou et al. [22] looked at error rates in different parts of the operating system kernel, and found that device drivers had error rates up to seven times higher than the rest of the kernel. Ozment and Schechter [23] examined the age of code as a predictor of vulnerability in the OpenBSD [29] operating system, and generally confirmed the finding that the rate at which bugs are found goes down over time. In a recent prior work [3], Li et al. directly compared both of these metrics to our popular paths metric, and found that neither was as predictive of vulnerabilities in the Linux kernel as our popular paths metric.

Shin et al. [24] examined code churn, complexity, and developer activity metrics, finding that these metrics together could identify around 70% of known vulnerabilities in two large open source projects codebase they studied. However, file granularity is too coarse to be useful when deciding which parts of the kernel need protection. Customizing the operating system kernel to work at the lines-of-code level, as the popular paths metric does, can be more effective.

Operating System kernel security enhancement.

Prior research in enhancing kernel security requires refactoring or modifying the kernel. Engler et al. [25] introduced the Exokernel architecture, which allowed applications to directly manage hardware resources, in the hope of gaining efficiency in accessing the hardware; this style of operating system design came to be known as a library OS. More recently, but in a similar spirit, unikernels, such as Mirage [26] run each application as its own operating system inside of a virtual machine, customizing the "kernel" for each application. Each of these systems, however, requires significant changes to existing applications (e.g., in the case of Mirage, applications must be rewritten in the OCaml programming language [30].) Other library OSes are written to run existing applications. Drawbridge [27] and Graphene [28] support unmodified Windows and Linux applications, respectively, by implementing an OS "personality" as a support library in each address space to improve security. However, user-space reimplementation of OS functionality incurs a performance overhead. Containers can mostly avoid this overhead, since they allow contained applications to make system calls directly. Thus, our work focused on the container-based approach.

Container security enhancement.

Existing security mechanisms available for containers usually leverage host kernel features. Namespaces is one mechanism that can provide isolation between processes, and has been adopted by Docker. Linux capabilities allow users to define and choose smaller groups of root privileges, and thus can help prevent containers from having risky privileges. Docker containers use Linux capabilities, and LinuxKit allows users to define and control the capabilities (such as file permissions) they want to use. While these existing approaches do help improve security for containers, they also have limitations. First, it can be really hard to understand how to correctly configure the security settings for containers. In fact, many users just use the default configuration, which may not be the best choice if strong security is needed. Second, in order to run applications the user wanted, containers still have to allow access to risky code in the host kernel. Our work improves the container security by imposing a fine-grained control over access to the risky kernel code.

7 Conclusion

This is our conclusion.

References

[1] 451 Research 2017. Cloud-Enabling Technologies Market Monitor Report. Retrieved April, 2019 from https://451research.com/images/Marketing/press_releases/Application-container-market-will-reach-2-7bn-in-2020_final_graphic.pdf

[2] Docker 2019. Enterprise Container Platform for High Velocity Innovation. Retrieved April, 2019 from https://www.docker.com

[3] Docker Hub 2019. A Library and Community for Container Images. Retrieved April, 2019 from https://hub.docker.com

[4] Kubernetes 2019. An open-source system for automating deployment, scaling, and management of containerized applications. Retrieved April, 2019 from https://kubernetes.io

[5] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. 2017. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Santa Clara, CA, USA, 1–13.

[6] LinuxKit 2019. A toolkit for building secure, portable and lean operating systems for containers. Retrieved April, 2019 from https://github.com/linuxkit/linuxkit

[7] NVD 2019. National Vulnerability Database. Retrieved April, 2019 from https://nvd.nist.gov

[8] OS Level Virtualization 2019. Wikipedia. Retrieved April, 2019 from https://en.wikipedia.org/wiki/OS-level_virtualisation

[9] VirtualBox 2019. An x86 and AMD64/Intel64 virtualization. Retrieved April, 2019 from https://www.virtualbox.org

[10] VMWare Workstation 2019. Running multiple operating systems as virtual machines (VMs) on a single Linux or Windows PC. Retrieved April, 2019 from https://www.vmware.com/products/workstation-pro.html

[11] W.G. Wong 2016. What's the Difference Between Containers and Virtual Machines? Electronic Design. Retrieved April, 2019 from https://en.wikipedia.org/wiki/OS-level_virtualisation

Table 1. Evaluation of the CVE vulnerabilities for the Linux kernel

#	CVE ID	CVSS Score	Description	Detected in the LinuxKit Popular Paths
1	CVE-2019-10124	7.8	denial of service, in mm/memory-failure.c	✗
2	CVE-2019-9213	4.9	kernel NULL pointer dereferences, in mm/mmap.c	✗
3	CVE-2019-9003	7.8	use-after-free	✗
4	CVE-2019-8956	7.2	use-after-free	✗
5	CVE-2019-8912	7.2	use-after-free	✗
6	CVE-2019-7308	7.5	out-of-bounds speculation on pointer arithmetic	✗
7	CVE-2019-3701	7.1	privilege escalation	✗
8	CVE-2018-1000204	6.3	copy kernel heap pages to the userspace	✗
9	CVE-2018-1000200	4.9	NULL pointer dereference	✗
10	CVE-2018-1000026	6.8	denial of service	✗
11	CVE-2018-20511	2.1	privilege escalation	✗
12	CVE-2018-20169	7.2	mishandle size checks	✗
13	CVE-2018-18690	4.9	unchecked error condition	✗
14	CVE-2018-18445	7.2	out-of-bounds memory access	✗
15	CVE-2018-18281	4.6	improperly flush TLB before releasing pages	✗
16	CVE-2018-18021	3.6	denial of service	✗
17	CVE-2018-16862	2.1	the cleancache subsystem incorrectly clears an inode	✗
18	CVE-2018-16658	3.6	local attackers could read kernel memory	✗
19	CVE-2018-16276	7.2	privilege escalation	✗
20	CVE-2018-15594	2.1	spectre-v2 attacks against paravirtual guests	✓
21	CVE-2018-15572	2.1	userspace-userspace spectreRSB attacks	✓
22	CVE-2018-14646	4.9	NULL pointer dereference	✗
23	CVE-2018-14634	7.2	integer overflow, privilege escalation	✗
24	CVE-2018-14633	8.3	stack buffer overflow	✗
25	CVE-2018-14619	7.2	privilege escalation	✗
26	CVE-2018-13406	7.2	integer overflow	✗
27	CVE-2018-12904	4.4	privilege escalation	✗
28	CVE-2018-11508	2.1	local user could access kernel memory	✗
29	CVE-2018-11412	4.3	ext4 incorrectly allows external inodes for inline data	✗
30	CVE-2018-10940	4.9	incorrect bounds check allows kernel memory access	✗
31	CVE-2018-10881	4.9	denial of service	✗
32	CVE-2018-10880	7.1	denial of service	✗
33	CVE-2018-10879	6.1	use-after-free	✗
34	CVE-2018-10878	6.1	denial of service	✗
35	CVE-2018-10074	4.9	denial of service	✗
36	CVE-2018-10021	4.9	denial of service	✗
37	CVE-2018-8781	7.2	code execution in kernel space	✗
38	CVE-2018-6555	7.2	denial of service	✗
39	CVE-2018-6554	4.9	denial of service	✗
40	CVE-2018-5390	7.8	denial of service	✗
41	CVE-2018-1130	4.9	NULL pointer dereference	✗
42	CVE-2018-1120	3.5	denial of service	✓
43	CVE-2018-1118	2.1	kernel memory leakage	✗
44	CVE-2018-1068	7.2	write to kernel memory	✗
45	CVE-2017-1000410	5.0	leaking data in kernel address space	✗
46	CVE-2017-1000407	6.1	denial of service	✗
47	CVE-2017-1000405	6.9	overwrite read-only huge pages	✗
48	CVE-2017-18224	1.9	race condition, denial of service	✗
49	CVE-2017-18216	2.1	NULL pointer dereference, denial of service	✗
50	CVE-2015-5327	4.0	out-of-bounds memory read	✗