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

One of several reasons why containers, such as Docker and LinuxKit, are so widely used is because of the perceived isolation and security they provide against the potentially malicious or buggy user programs running inside of them. However, containers provide no protection from the zero-day kernel bugs inside the kernel itself, which, if triggered, could lead to such security problems as privilege escalation. Containers remain vulnerable because they allow access to rarely executed paths where such bugs can be found. Limiting kernel access to only frequently used ***popular paths***, which have previously been proven to contain fewer security bugs, would greatly reduce this risk. Therefore, if a container could run using only these paths, its security would be greatly enhanced.

In this paper, we propose a method to enhance the security of existing container designs by leveraging the popular paths metric. We designed and implemented a ***Secure Logging System***, which first systematically identifies popular paths data in widely-used containers, and uses this data to create a kernel that generates security logs and warning messages when a potentially risky path is reached. Users can then decide whether the potentially dangerous code execution should be stopped. We tested this modified kernel, which we named the ***Secure Logging Kernel***, to determine its ability to run containers. Our evaluation verified that, more than 99.9% of the time, containers were able to run their default workload using just the popular paths. Furthermore, only 6% of 50 Linux kernel CVE security vulnerabilities we examined were present in the popular paths on which these containers ran, greatly reducing any potential risk of triggering kernel bugs. Lastly, our performance evaluation shows that containers running our Secure Logging Kernel only incur about 0.1% runtime overhead on average, and around 0.37% extra cost of memory space. This means the Secure Logging Kernel can improve security with little or no 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 [17]. This widespread adoption of containers, such as Docker [19] and LinuxKit [24], can be attributed to several factors, including portability and elimination of the need for a separate operating system [29]. In addition, the perceived isolation and security they provide to the user programs running inside of them is reassuring to users looking to protect their data and operation. However, this perception may be false, as several recent incidents have demonstrated it is possible to trigger zero-day kernel bugs from inside of a container [21, 28]. Furthermore, since the kernel is the critical and privileged code shared by containers and the host system, exploitation of such bugs could lead to severe security problems, such as privilege escalation. The famous "Dirty COW" vulnerability that emerged in 2016, reported as CVE-2016-5195, allow attackers to escape from a Docker container and access files on the host system [15]. This vulnerability affected all of the Linux-based operating systems that used older versions of the Linux kernel, including Android. One analysis, conducted a year after it was first reported, found the bug in more than 1200 malicious Android apps, affecting users in at least 40 countries [16].

The fundamental reason such exploits can occur is that existing containers are designed to directly access the underlying host operating system kernel on which they are run. This design, which makes containers more efficient and lightweight to use—features that make them so attractive to end-users— unfortunately exposes them to zero-day bugs within the kernel itself [28]. To this point, there has been little or no effort to limit such contact, in part because there was no efficient way to identify where these bugs might be. If one knew which part of the kernel was free of 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.

Yet, targeting where the bugs are likely to be has been an elusive goal for many years. A number of researchers have promoted metrics [18, 27] that claim to be able to predict the presence of bugs, but most have proven only minimally effective. That is, until two years ago when a study [23] demonstrated a powerful correlation between the popularity of a kernel code path and its likelihood to contain a security flaw. Furthermore, the study [23] demonstrated that these frequently used kernel paths can be leveraged to design and construct 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? To do so, we would first have to answer

the question: Can applications in containers run without access to the whole kernel?

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 first, finding a way to identify the parts of the underlying kernel that are less likely to contain security bugs. We were able to demonstrate a systematic way to gather data on popular paths, and affirmed that this data is representative of hundreds of popular Docker containers from Docker Hub [20]. Next, we produced a Secure Logging Kernel designed to log any attempt to access the unpopular paths, and to warn users away from code found in the less-used paths. The Secure Logging Kernel contributed to our study in two ways. Firstly, it provided usable data on how often applications used these risky paths, so we could determine whether such paths are essential to their functionality. And, secondly, it built into our instantiation of the modified kernel a way for users to detect potentially dangerous program behaviors and make decisions about whether or not to block executions in unpopular paths.

An evaluation of our findings affirmed that popular Docker containers experience no loss of functionality when using just the popular paths. Indeed, for the official Docker containers' default workload, the popular paths were used more than 99.9% of the time. Our security evaluation confirmed that using the popular paths to run Docker containers can effectively prevent most kernel bugs from being triggered, as only three of the 50 CVE kernel security vulnerabilities we checked for were found in those LinuxKit paths. And, in our performance evaluation, we found that running our Secure Logging Kernel only incurred about 1% of runtime overhead, while the memory space overhead was only about 0.37%. Thus, greater security could be achieved with very little perceived difference in operation efficiency or cost.

In summary, we make the following contributions in this paper:

- We systematically identify and capture the "popular paths" data from Docker containers running in the LinuxKit VM.
- We design and implement a Secure Logging System, that utilizes the popular paths data to create a modified Linux kernel (Secure Logging Kernel). This instrumented kernel outputs security warning messages whenever there is an attempt to reach and trigger unpopular paths.
- Using the Secure Logging Kernel, we demonstrate that popular Docker containers were able to run their default workload normally using the popular paths more than 99.9% of the time, with negligible (less than 1%) performance overhead.

2 Background and Motivation

The concept of full virtualization, where a guest operating system runs programs in isolation, has been around for several decades [26]. Yet, the emergence of container programs [32], which directly utilize the host kernel code to perform OS functionality, and therefore have a lower overhead than full virtual machines, has spurred growth in container popularity. Docker [19], LinuxKit [24], and Kubernetes [22] have all seen wide adoption over the past few years. For example, Docker Hub [20] now 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 [31] and VirtualBox [30], 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 sufficiently secured them against potential threats.

This perception has been bolstered over the years by the deployment of a number of security and isolation mechanisms designed to protect the programs running inside of them. Linux introduced Namespaces [9] to the kernel between versions 2.6.15 and 2.6.26, to provide a global system resource abstraction that helps achieve isolation between different containers. Docker and LXC [11] containers also use this technique. Another key Linux feature called control groups [7] can be used to limit, account for, and isolate important system resources, such as memory, CPU, and disk I/O, to each container. This feature not only controls resources, but also helps prevent denial-of-service attacks.

When Linux kernel capabilities [8] were introduced, container systems, such as Docker were also able to restrict permissions on specific resources, such as network and file system. Another option for enhancing kernel security has emerged, with the development of kernel hardening systems, such as AppArmor [1], SELinux [14], GRSEC [5], and PAX [12]. Acknowledging that buggy code is likely unavoidable, these systems look to make the kernel code resilient enough to attacks that any resident bugs cannot be exploited. 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 [13]. These kernel hardening systems are mostly access control mechanisms that provide safety in a way similar to that of capabilities.

These existing mechanisms do improve security to a certain degree, but do not address the aforementioned access problem. Unlike in a full-scale virtual machine, the host kernel is shared among all the containers and the host operating system. This magnifies the damage a kernel vulnerability could cause, in addition to, breaking the isolation that users expect from using a container.

The design initiative described in the next section was launched specifically to enhance container security by preventing zero-day kernel bugs from being triggered inside containers, while still allowing containers to perform their assigned workload. Having already established in a previous paper that leveraging the popular paths metric could identify and neutralize the threat of these bugs [23], we proposed that container security can be enhanced in the same manner. To prove this point, we designed and implemented the Secure Logging System.

3 Design and Implementation

Designing an implementation of the popular paths metric for use with containers required completion of two separate tasks. First, we had to find the popular paths for the underlying host operating system. And, second, once we knew where these paths were, we needed a way to log / block the unpopular paths to warn and keep containers away from these less-used and potentially buggy code paths. Our solution was a dual module approach we call the Secure Logging System (Shown in Figure 1).

3.1 Design of Our "Secure Logging System"

Identifying the popular paths for the given container system is handled by the system module we refer to as the Kernel Profiler. This module identifies the lines of code in the host kernel that are executed when running its default / regular workload. Such a workload is often defined in the configuration files (Dockerfiles) that come with the container images. We collected "popular paths kernel traces," as we named them, from a set of the containers, as ranked "most popular" by the number of user downloads.

In effect, the Profiler sets out a map highlighting places in the kernel that are safe for an application to access. The second module, the Kernel Modifier, uses this map to instrument the rarely used unpopular paths to either generate security warning messages / logs, or to block code execution on these paths. When both modules are implemented, the result is an instrumented Linux kernel with security monitors and checks inserted at the non-popular paths. This Secure Logging Kernel can then be used to run containers without any required changes to the containers themselves.

3.2 Implementation

To adapt our design for real-world containers and applications, it was necessary to build our own infrastructure. This section describes in detail how the modules in our system were implemented.

3.2.1 Implementing the Kernel Profiler

The Kernel Profiler is designed to collect the kernel trace of running user containers. The popular paths kernel trace we are looking to identify refers to the lines of code in the underlying host operating system kernel that were executed when running the regular container workload. This workload is defined in the configuration files of the corresponding images from containers we labeled popular based on the number of user downloads. Our Kernel Profiler works in the following way:

1. The Linux kernel used to run the Docker containers is recompiled with the Gcov [4] kernel profiling feature enabled.
2. To run the Docker containers, the Kernel Profiler first automatically generates the configuration file for the Gcov-enabled LinuxKit. This file will define the task container, along with a data container responsible for collecting, storing, and transferring the kernel trace data.
3. When booting the LinuxKit virtual machine, a script in the profiler automatically starts running the workload of the task container. The profiler will collect the kernel trace data, in the form of gcda files generated by Gcov, store them at "/sys/kernel/debug/gcov/," and by the end of the run transfer the data to the host system for further use.
4. Using the lcov [6] tool to process the kernel trace data collected from the host system, the profiler generates formatted data about which lines of code were executed in the kernel source files. These files will be used by the Kernel Modifier to identify the unpopular paths that must trigger an alert if used.

3.2.2 Implementing the Kernel Modifier

For the Kernel Modifier to insert security logging code at the unpopular paths in the Linux kernel, It needs to first identify the correct places in the kernel source files to be instrumented. It can then modify the code as needed. The Kernel Modifier has three main parts: the Clang compiler, the Clang analyzer, and the kernel instrumenting tool (Shown in Figure 3). It operates in the following way:

1. The Clang C compiler from the Clang / LLVM project [10], along with a BEAR [2] tool compiled the Linux kernel source. Using BEAR, we can generate a compilation database containing all the compile flags and options needed for the process. In turn, this database will be used by our Clang analyzer to perform source code parsing in the next step.
2. Once we have the compilation database for the Linux kernel, we use the Clang analyzer to perform static analysis on the kernel source code to obtain the control flow graph for each function in the files. The analyzer leveraged Clang's LibTooling [3] to construct the AST tree from the kernel source code, and then obtain the control flow graph. With this graph, we can identify the corresponding source line number for each basic block. Furthermore, the beginning lines of all the

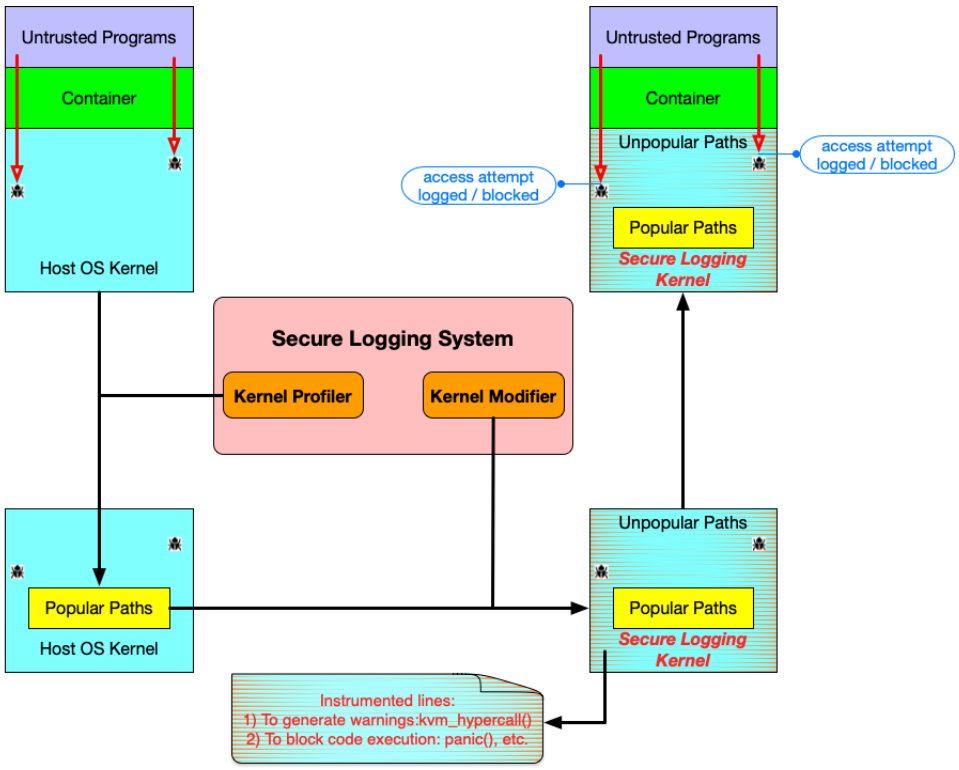

Figure 1. Design overview of our Secure Logging System

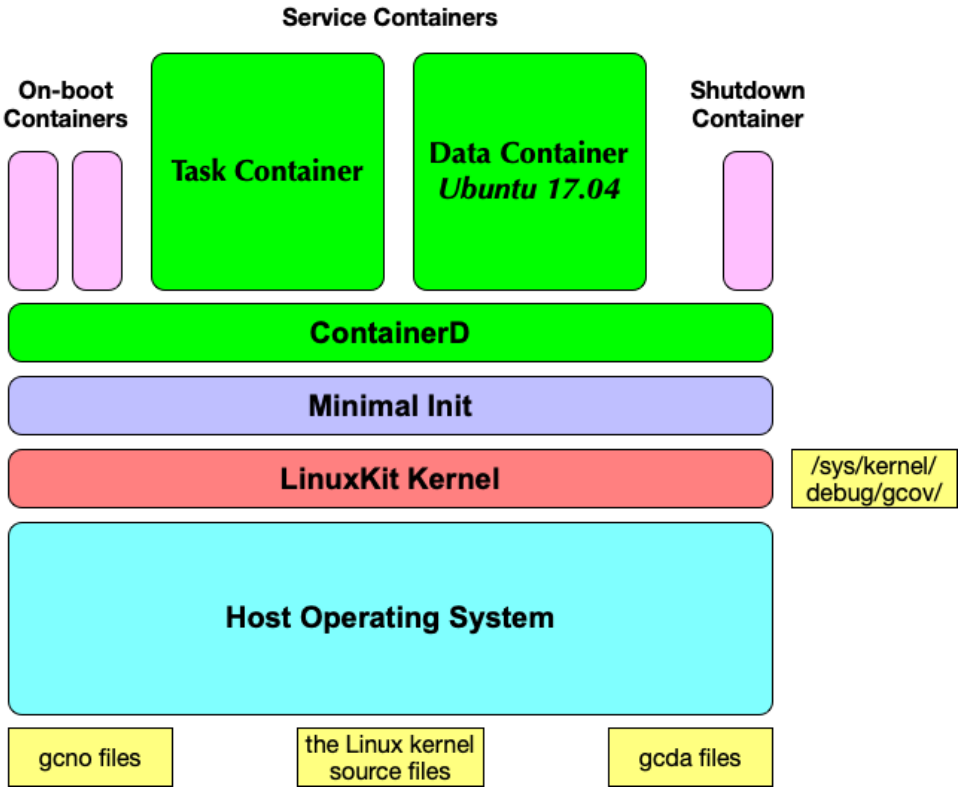

Figure 2. Implementation of the Kernel Profiler

blocks in the source files are candidates for places to add security logging code. Here, a basic block refers to a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. Thus, if a basic block is unpopular, it is sufficient to insert only one piece of secure logging code at the beginning of it to monitor and warn the attempt to reach that code. This approach can optimize our instrumentation by avoiding redundancy in our code injection.

3. The next steps has the kernel instrumenting tool executing modifications based on both the basic blocks, and the collected popular paths data . The algorithm for our kernel modification directs us through the basic blocks in the kernel source files. If none of the lines in a block are in the popular paths data, we consider this an unpopular basic block, and our instrumenting tool will add the security logging code at the beginning of this block. If we discover that an entire function has no lines in the popular paths data, we consider this an unpopular function, and just add our security logging code once where it starts . This allows us to avoid adding redundant and unnecessary code. The secure logging code we inserted in front of the unpopular paths was a kvm hypercall from the LinuxKit kernel into the host Linux kernel. In this way, we can guarantee minimal affect on the LinuxKit kernel functionality, while still being able to generate security logging whenever unpopular paths were reached.
4. Our Kernel Modifier automatically inserts our secure logging code at beginning of the unpopular paths in the Linux kernel source files, and then recompiles the kernel to produce the Secure Logging Kernel, which 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 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.

Table 1. LTP tests

	Original Linux kernel	Secure Logging Kernel
Total Tests	798	798
Total Skipped Tests	20	20
Total Failures	105	105

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 2). 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 [25]. 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.

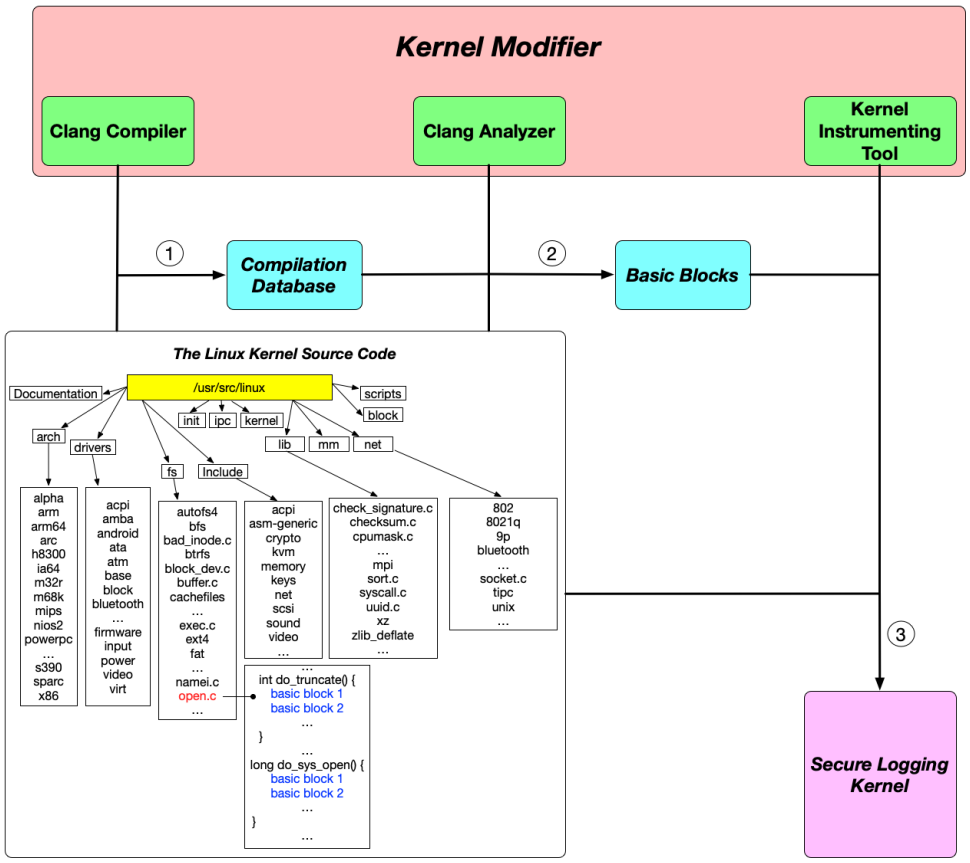

Figure 3. Implementation of the Kernel Modifier

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 2). 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 Limitations

While our results and analysis indicate that our dataset was representative of the common workload users performed when using container applications, we believe that a larger sample base with more runs could potentially capture the race-condition related kernel footprint more comprehensively, thus rendering more accurate popular paths data.

Another factor that could have affected our work was the type of images we used. We limited our selection to official container images from Docker Hub, while there are a number of popular Docker images from open source projects on Github that we did not access. With our methodology used in this work, running these additional containers to obtain and analyze their popular paths might have given us a broader spectrum of results.

Lastly, in this work, we ran our system and produced results with Linux kernel version 4.14.24. For future work, we plan to extend our system so that it can automatically work with different kernel versions. This will generalize the use of our work, and potentially help a lot more users to secure their containers.

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] [n. d.]. AppArmor. https://wiki.ubuntu.com/AppArmor

[2] [n. d.]. Build EAR. https://github.com/rizsotto/Bear

[3] [n. d.]. Clang LibTooling. https://clang.llvm.org/docs/LibTooling.html

[4] [n. d.]. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[5] [n. d.]. GRSEC. https://wiki.debian.org/grsecurity

[6] [n. d.]. Lcov. http://ltp.sourceforge.net/coverage/lcov.php

[7] [n. d.]. Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html

[8] [n. d.]. Linux kernel capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html

[9] [n. d.]. Linux Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html

[10] [n. d.]. The LLVM Compiler Infrastructure Project. https://llvm.org

[11] [n. d.]. LXC. https://linuxcontainers.org

[12] [n. d.]. PAX. https://pax.grsecurity.net

[13] [n. d.]. Red Hat Linux. https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

[14] [n. d.]. SELinux. https://selinuxproject.org/page/Main_Page

[15] 2016. Dirty COW - (CVE-2016-5195) - Docker Container Escape. https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/

[16] 2017. Dirty Cow vulnerability discovered in Android malware campaign for the first time. https://www.zdnet.com/article/dirty-cow-vulnerability-discovered-in-android-malware-campaign-for-the-first-time/

[17] 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

[18] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 73–88. https://doi.org/10.1145/502034.502042

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

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

[21] Sergiu Gatlan. 2019. RunC Vulnerability Gives Attackers Root Access on Docker, Kubernetes Hosts. https://www.bleepingcomputer.com/news/security/runc-vulnerability-gives-attackers-root-access-on-docker-kubernetes-hosts/

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

[23] 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.

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

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

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

[27] Andy Ozment and Stuart E. Schechter. 2006. Milk or Wine: Does Software Security Improve with Age?. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA, Article 7. http://dl.acm.org/citation.cfm?id=1267336.1267343

[28] Vijay Pandurangan. 2016. Linux kernel bug delivers corrupt TCP/IP data to Mesos, Kubernetes, Docker containers. https://tech.vijayp.ca/linux-kernel-bug-delivers-corrupt-tcp-ip-data-to-mesos-kubernetes-docker-containe

[29] P. Rubens. 2017. What are containers and why do you need them? https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.htm

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

[31] 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

[32] 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 2. 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	✗