健康資訊交換系統中之容器安全

The Container Security in Healthcare Data Exchange System

國立中山大學資訊工程學系

Department of Computer Science and Engineering

National Sun-Yet-San University, Taiwan

110 學年度大學部專題製作競賽

Bachelor's degree graduation project in 2021

Author: Chih-Hsuan Yang (B073040047)

Advisor: Chun-I Fan

October 18, 2021

Abstract

This research proposes a mechanism to enforce the system call a specific policy in the container, which is deployed in runtime. This policy is designed for the FHIR healthcare data exchange standard's container, which could guarantee the FHIR server does not have unsupported behavior and takes almost zero overhead. Recently, many companies use containers to run their microservices since containers could make their hardware resources be used efficiently. And the newest healthcare data exchange standard FHIR (Fast Healthcare Interoperability Resources) [1] has been implemented in a container by IBM, Microsoft, and Firebase. The deployment of FHIR in a container is a trend in the digital world [1]. However, containers are not sandboxes [2] . Containers are just isolated processes [2]. Therefore, if hackers or malicious software could sneak into the container that would be a new cyber attacking surface in nearly future.

[1]FHIR official: https://www.hl7.org/fhir/

[2]gVisor GitHub: `https://github.com/google/gvisor`

Contents

Chapter 1

Introduction

1.1 Container and Linux Kernel

The container is a secondary product of the operating system in the past 20 years. The FreeBSD develops 'Jails' in 1999, and the Solaris develops 'Zones' in 2004. Linux also took this idea into the Linux kernel, which is named cgroups (2007), the capabilities (2003), and seccomp (2005). However, why the Linux breaks this technology into many parts? This is because they had discussed: "Why Should a System Administrator Upgrade?" in 2001 [1]. The Linux kernel almost entered the development path of "upgrade for demand" like Microsoft Windows, and deviated from the original path of "providing a mechanism but not a strategy" of the original Linux kernel.

While Linux were spreading in various server or distributed system, the Linux community got more pull requests to solved the scalability and virtualization issues [3]. However, they avoided confusion caused by multiple meanings of the term "container" in the Linux kernel context. In kernel version 2.6.24 (2007) [2], control groups functionality was merged into the mainline, which is designed for an administrator (or administrative daemon) to organize processes into hierarchies of containers; each hierarchy is managed by a subsystem. Moreover, the cgroups was rewrote into cgroups-v2 in Linux kernel 4.5 (2015) [3].

The first and most complete implementation of the Linux container manager was LXC (Linux Containers). It was implemented in 2008 using cgroups and namespaces, and it runs on a single Linux

[1] Version 2.4 of the LINUX KERNEL–Why Should a System Administrator Upgrade? https://www.informit.com/articles/article.aspx?p=20667

[2] Notes from a container: https://lwn.net/Articles/256389/

[3] Control Group v2: https://www.kernel.org/doc/Documentation/cgroup-v2.txt

kernel without requiring any patches. LXC provides a new view and imagination of virtualized services without any hypervisor. In 2016, Docker replaced LXC with ”libcontainer”, which was written in the Go programming language. Docker combined features in a new, more attractive way and made Linux containers popular.

The secondary product of the operating system, containers, offering many advantages: they enable you to ”build once, run anywhere.” Docker does this by bundling applications with all their dependencies into one package and isolating applications from the rest of the machine on which they’re running. Therefore, this research is based on docker container to propose a scheme of healthcare data exchange system’s security.

1.2 FHIR

FHIR is a standard for healthcare data exchange. The FHIR standard will be used in Taiwan in the near future. FHIR will be used to provide PHR (Personal Healthcare Records) in Taiwan. Therefore, we choose the most popular standard ”FHIR” for the target of the healthcare data exchange system.

1.2.1 RESTful API and Data Structure

REST (Representational State Transfer) is a stateless reliable web API, which is based on HTTP methods to access resources or data via URL parameters and the use of JSON or XML format to transmit queries. Because the RESTful is stateless, the client should keep their information (i.e. cookies) by themself.

FHIR has features: RESTful and data structure, make our research and benchmarks more accurate and reliable. Statelessness is a developer-friendly feature, the developer and the tester would not to design a complex state machine on the server-side or generating test files. And the FHIR takes RESTful as standard. Moreover, FHIR standard declared the ‘StructureDefinition’ [4]. These structure definitions are used to describe both the content defined in the FHIR specification itself - Resources, data types, the underlying infrastructural types, and also are used to describe how these structures are used in implementations.

[4] FHIR Resource Structure Definition: http://www.hl7.org/fhir/structuredefinition.html

1.2.2 Why IBM FHIR server

There are many applications using IBM's FHIR server as the base component of the EHR (Electronic Health Records) system to communicate with the other various databases. Take it for example that the NextCloud's EHR service, Taipei Veterans General Hospital, and AWS Cloud are using the FHIR server in a container for subroutine service.

NextCloud is an open-source and self-hosted productivity platform for users. Many people caring about their privacy issues distrust the FAANG (Facebook, Amazon, Apple, Netflix, Google), so they are using NextCloud to keep their privacy on their own. Therefore, they are eager to have a secure EHR system for their PHR [5].

1.3 Data and Privacy

TBD...

[5] Richard Stallman talks about IoT

Chapter 2

Related Work

2.1 Collecting System Calls

There are several pieces of research to detect intrusions or unexpected behaviors by collecting the system calls methods in runtime [4, 5, 6, 7]. Amr S. Abed et al.[4] proposed a real-time host-based intrusion detection system in a container, which is based on system call monitoring. They use the 'strace' command to collect a behavior log to a system call parser. Then use the BoSC (Bag of System Calls) [8] to classify is it a normal behavior in the database.

The BoSC technique is a frequency-based detection tip. Kang et al.[8] defined those distinct system calls in $\{c_1, c_2, \ldots, c_n\}$, For all system call s_i had been called in c_i times. And they use Naïve Bayes classification to deduce if it is unexpected behavior. Then the Amr S. Abed et al. give the false positive rate around 2% in $O(S + n_k)$ epochs to the MySQL database [4].

- Epoch Size (S): The total number of system calls in one epoch.
- n_k: It is the size of the database after epoch k.

However, the BoSC is running in user space, even though it is a background service running on the same host kernel. It might have heavy constant time costs of copying data from user to kernel and kernel to user by the 'copy_to_user()' and 'copy_from_user()' calls.

Mohamed Azab et al.[6, 7] takes a mathematical model to simulate the smart moving target defense for Linux container resiliency. Considering an 'ESCAPE' model is the interaction between attackers and their target containers as a "predator searching for a prey" search game. This search game has 3 modules: behavior monitoring, the checkpoint/restore, and the live migration modules. This model is running on the same host and the same attacking surface because they considered the containers (prey) are running on the same machine with some migration probability.

They show the survival rate in Amr S. Abed et al.[4] model for some zero-day vulnerabilities in different types and numbers machines. Mohamed Azab et al. [6, 7] concluded that an IDS could detect and avoid mobile continually-growing attacks efficiently by the 'ESCAPE' model with collecting system calls.

2.2 Fine-grained Permission Control

2.2.1 Capabilities

There are 49 different capabilities in today's Linux kernel 5.13 [1]. A capability can be assigned to a task (i.e thread or process) to determine if the task can use the fine-granted system calls. For example, we give a thread CAP_SYS_BOOT, then the thread can use the reboot [2] and the kexec_load [3] system call.

Xin Lin et al.[9] collected 27 CVE vulnerabilities that could cause the privilege escalation attacks. There are only three vulnerabilities that could bypass the capabilities protection of the Linux kernel. And the other 24 escalation vulnerabilities, could be filtered by the fine-grained permission control with capabilities. Those three (CVE-2016-8655, CVE-2017-5123, CVE-2017-7308) bypassed capabilities vulnerabilities are attacking kernel-level race conditions.

The CVE-2016-8655[4] is a bug in net/packet/af_packet.c. We often use the CAP_NET_RAW namespace in the container to make unprivileged users be able to use some privileged net-util commands. The bug is that there exists a race condition probability to race the unauthorized data inside packet_set_ring() and packet_setsockopt()[5] such that there is a chance to modify the socket version to TPACKET_V_1 before the packet_set_ring function. However, it would be 'kfree' the timer in the TPACKET_V_1. Then we can take the timer, which is used after free, to control the SLAB adopter to write the st_uid by itself [6].

[1] https://man7.org/linux/man-pages/man7/capabilities.7.html
[2] https://man7.org/linux/man-pages/man2/reboot.2.html
[3] https://man7.org/linux/man-pages/man2/kexec_load.2.html
[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655
[5] Linux af_packet.c race condition (local root) https://lwn.net/Articles/708319/
[6] CVE-2016-8655 PoC: https://www.exploit-db.com/exploits/40871

2.2.2 Linux Secure Module

There were many pieces of research designing great rules of the LSM (Linux secure module). LSM is a mandatory access control framework in kernel, which is supported from Linux 2.6 (2004). The AppArmor[7] and SELinux[8] are common LSM in the kernel. All the Android devices are fully enforced the SELinux after Android 5.0. [9]

The SIDs (Security IDs) and permissions, which are the identifiers for access control policies, make up the security policy of SELinux or AppArmor. Files and directories, which are the actual protected objects for the SID, are mapped to the SELinux or AppArmor of each system [10, 11, 12]. The SELinux is much more incredibly complex than AppArmor, but with this complexity, you have more control over how tasks are isolated. However, the AppArmor is so straightforward that people can write the configuration profile by themself.

SUNG-HWA HAN et al.[13] had proposed an architecture to enforce the access control of image's layers. Because the docker engine does not guarantee the layers could not be modified by the host environment. Therefore, if we give a container privileged permission, it could modify the layers of images. The research [13] is using the LSM's policy table to enforce the access control of the file system in the kernel.

This paper [13] shows that the performance is almost the same as raw SELinux, and the branch costs of indexing the policy table could be a constant value in the CPU rate measurement. However, the research is only be measured in overlay2 file system. SUNG-HWA HAN et al. said if it has the same performance in AUFS or the other file systems, the above results could be more reliable.

Yuqiong Sun et al.[14] proposed separate the security namespace. Each container can route their operation to different security namespaces for their "comment". Each involved in the security namespace independently makes a security decision, and the operation is allowed only if all the secure namespaces allow.

The policy engine and the security namespace are implemented in . . .

[7] https://apparmor.net/
[8] https://github.com/SELinuxProject/selinux
[9] https://source.android.com/security/selinux#supporting_documentation

2.3 Recently Exploited Vulnerabilities

In this section, we will mention and review some ‘High’ or ‘Critical’ vulnerabilities in CVSS (Common Vulnerability Scoring System).

TBD...

2.3.1 Five Stages of Malware

We had been inspired by the quark engine[10], which is an open-source malware scoring system for Android APK files. The quark engine had been developed from the Taiwan Criminal Law’s five stages: (i) determination, (ii) conspiracy, (iii) preparation, (iv) start, (v) practice.

We also can use these five stages to give the malware limitation to exploit the vulnerabilities.

TBD...

2.3.2 Case studies

The Dirty CoW

Delwar Alam, et al.[15] showed the race condition and the mechanism of ”Copy on Write”. ”Copy on Write” is a resource-management technique used in computer programming to efficiently implement a ”duplicate” or ”copy” operation on modifiable resources[16]. It is often inspired when ‘fork‘ or ‘mmap‘.

Mechanism TBD...

CVE-2016-8655 series

TBD...

2.4 Virtual Environment Performance Benchmark

There is a trend of applications are developed or deployed into microservice in a virtual environment since 2008. And the performance benchmark of applications in the virtual environment becomes more and more critical.

[10] https://quark-engine.readthedocs.io/en/latest/

Therefore, there are many pieces of research shows how to evaluate the performance when using containers or the other virtual infrastructures[17, 18, 19, 20]. They are comparing the throughput, latency, and QoS for memory IO, or cryptography algorithms calculating costs.

Ethan G. Young, et al.[20] showed the gVisor costs: $2.2\times$ system call overhead, $2.5\times$ memory allocation latency, and $216\times$ **slower** than raw system on complex file opening.

TBD...

Chapter 3

Preliminary

3.1 Container's Components

3.1.1 Namespaces

3.1.2 Cgroups

3.1.3 Seccomp

3.2 Programs in Execution

3.2.1 The task_struct in Kernel

3.2.2 Capabilities

3.3 Sandbox Security

3.3.1 User Mode Linux

gVisor

3.3.2 Virtual Machines

3.4 The (e)BPF

Chapter 4

Proposed Scheme

4.1 Workflow

In short, our proposal is generating a perfectly fittable mask layer which is coupled with the healthcare data exchange system in build time.

We proposed a CI/CD workflow to guarantee the runtime enforcement of policies in figure 4.1. Each block of the workflow will be described in the following subsection.

Because of the CI/CD workflow, we can rolling update all the features or fixing vulnerabilities, such that, the software would be released secure eventually. Linus Torvalds said[1] : ”The only real solution to security is to admit that bugs happen, and then mitigate them by having multiple layers.” And our layer is enforced in kernel space, therefore, there are no existing other attacks that can be inflected in the user program except for the kernel exploit.

[1] https://www.youtube.com/watch?v=5CIL54-KKz0

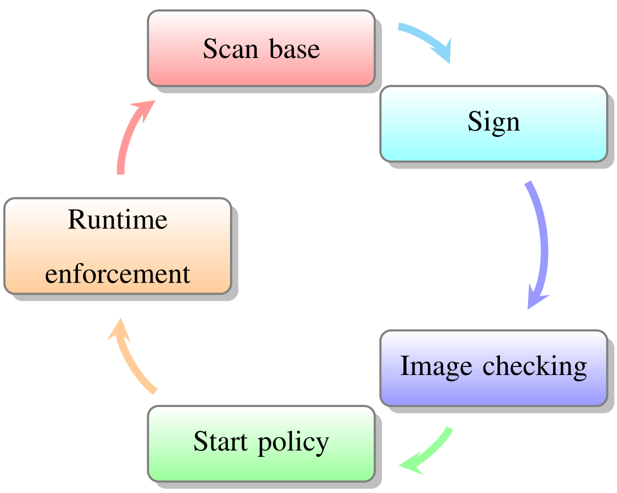

Figure 4.1: Contiguous Integration and Contiguous Deployment

4.1.1 Scan Base Image

We scan all the layers which construct the image of the container recursively. All containers are images in execution, that is we can treat the container as an image in runtime. Therefore, the layers of image construction have to be trusted.

For a general image I_i which has been constructed in n layers L_i, $\forall i \leq n$, $n \in \mathbb{N}$, we can use the spotbugs[2] or the other bug-scanning tools to ensure that the software is a bugless program. The bugless program p_i is in the layer L_i which construct the I_i

4.1.2 Building and Signing

We will execute the developer's unit tests and the integration test in the build time. We catch all the system calls s_i by the BoSC[8] method, and generate the $S = \{s_1, s_2, \ldots s_i \ldots s_n\}$ set from the program's n system calls, $S \subseteq \mathbb{S}$, the $\mathbb{S}$ is all the system calls that the kernel supported. We wrote a driver to parse the S into a whitelist filter of seccomp's policy P.

Through the workflow above, the L_i's security is almost surely enough. Then we sign our certificate C and the policy R to the image I_i, which is constructed by those trusted layers L_i into $\hat{I}_i$. That is $\hat{I}_i = C(P \oplus \Sigma_{\forall i} L_i)$.

4.1.3 Check Image and Policy

When we deploy the $\hat{I}_i$ into an active machine, we have to check the C of $\hat{I}_i$ is valid for signer's trusted verification server.

The verification server can check the certificate C's integrity and encrypt those checking results by the server's private key P_{VK} to the active machine. The active machine will also check the certificate C' from the verification server bidirectionally.

And we register our policy P into the active machine's kernel to limit the $\hat{I}_i$ launched by the user in runtime, that is the container.

4.1.4 Enforce the Policy

The kernel of the active machine can help us to guarantee the policy P is enforced in kernel space. Since the container is launched by the user, the policy P has been invoked in each system calls of the container. Because the policy P is a whitelist, all of the other system calls which do not belong to the signed container's application would send a permission denied signal from the kernel.

[2] https://spotbugs.github.io/

4.2 Rolling Updates

The rolling update is a trend of software engineering products, which is also named agile software development. Eric S. Raymond formulated the Linus's law in *The Cathedral and the Bazaar*[21]. We give enough eyeballs and layers, all bugs or vulnerabilities are shallow in our healthcare data exchange system. Therefore the container can be secure eventually.

Chapter 5

Analysis and Benchmark

5.1 Analysis

5.1.1 Attacking Surface

5.1.2 Time Consuming

5.1.3 Statistics

Figure 5.1.3 is the FHIR server's all system calls in BoSC[8] and the number of called times.

5.2 Benchmark

5.2.1 Latency

Figure 5.2.1 is the concurrent processes transporting time difference in container and virtual machine.

5.2.2 Throughput

TBD...

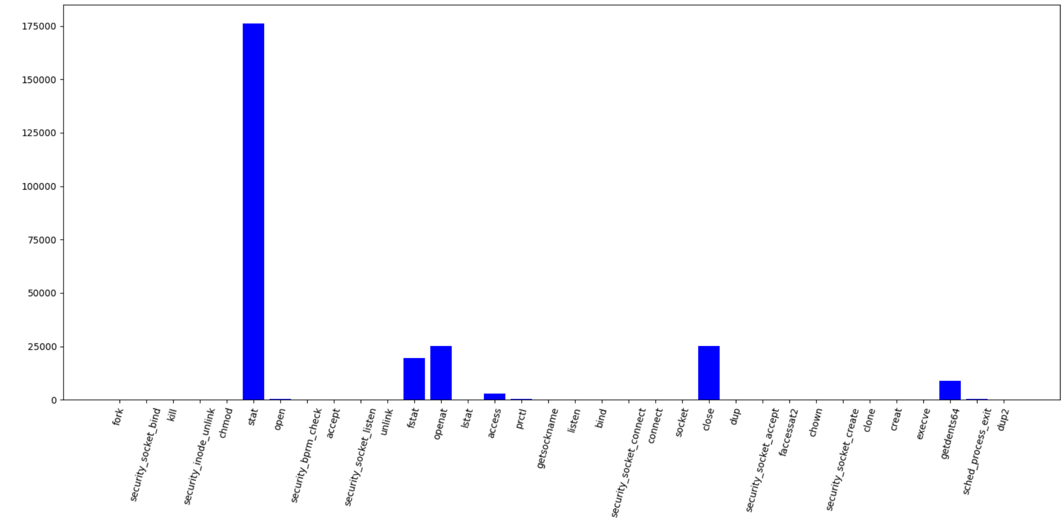

Figure 5.1: All the system calls which the FHIR called times

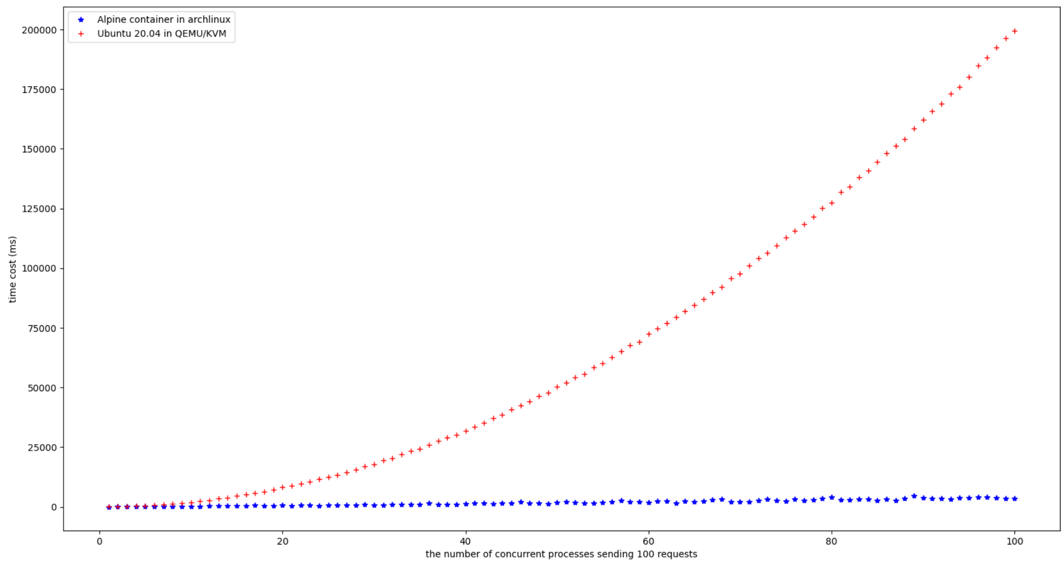

Figure 5.2: Concurrent processes transporting time

Chapter 6

Conclusion

We can see the comparison results in virtual machine and container are significantly indifferent order of time-consuming. There is no exist the gVisor's result is because the gVisor was not able to launch the IBM/FHIR server system, which is the target in our research. We also expect the gVisor might run faster significantly than the virtual machine, however, our target cannot be launched successfully in gVisor's sandbox.

We thought there might have been some race condition bugs via JWE(JAVA Web Engine) in gVisor. We found the IBM/FHIR server return an error code 141 while it launching. However, we did the same configuration in Docker with our policy and raw gVisor. Therefore, we thought the gVisor did not do well to supports all system calls.

And the time complexity of the virtual machine is significantly different from the container. We propose a hypothesis of the time complexity of the virtual machine, because there are more page fault events and limited by the throughput of virtual machine device driver[2, 19].

6.1 Better Architecture

6.2 Future Machine Learning in Kernel

Reference

[1] Arif Ahmed and Guillaume Pierre. "Docker Container Deployment in Fog Computing Infrastructures". In: *2018 IEEE International Conference on Edge Computing (EDGE)*. 2018, pp. 1–8. DOI: 10.1109/EDGE.2018.00008.

[2] Ian Goldberg et al. "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker". In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*. SSYM'96. San Jose, California: USENIX Association, 1996, p. 1.

[3] Silas Boyd-Wickizer et al. "An Analysis of Linux Scalability to Many Cores". In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Oct. 2010. URL: https://www.usenix.org/conference/osdi10/analysis-linux-scalability-many-cores.

[4] Amr Abed, Charles Clancy, and David Levy. "Intrusion Detection System for Applications Using Linux Containers". In: vol. 9331. Sept. 2015, pp. 123–135. ISBN: 978-3-319-24857-8. DOI: 10.1007/978-3-319-24858-5_8.

[5] José Flora. "Improving the Security of Microservice Systems by Detecting and Tolerating Intrusions". In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2020, pp. 131–134. DOI: 10.1109/ISSREW51248.2020.00051.

[6] Mohamed Azab et al. "Smart Moving Target Defense for Linux Container Resiliency". In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. 2016, pp. 122–130. DOI: 10.1109/CIC.2016.028.

[7] Mohamed Azab et al. "Toward Smart Moving Target Defense for Linux Container Resiliency". In: *2016 IEEE 41st Conference on Local Computer Networks (LCN)*. 2016, pp. 619–622. DOI: 10.1109/LCN.2016.106.

[8] Dae-Ki Kang, D. Fuller, and V. Honavar. "Learning classifiers for misuse and anomaly detection using a bag of system calls representation". In: *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*. 2005, pp. 118–125. DOI: 10.1109/IAW.2005.1495942.

[9] Xin Lin et al. "A Measurement Study on Linux Container Security: Attacks and Countermeasures". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 418–429. ISBN: 9781450365697. DOI: 10.1145/3274694.3274720. URL: https://doi.org/10.1145/3274694.3274720.

[10] Stephen Dale Smalley, Chris Vance, and Wayne Salamon. "Implementing SELinux as a Linux Security Module". In: 2003.

[11] Doug Kilpatrick, Wayne Salamon, and Chris Vance. "Securing The X Window System With SELinux". In: 2003.

[12] Luis Franco, Tony Sahama, and Peter Croll. "Security Enhanced Linux to enforce Mandatory Access Control in Health Information Systems". In: *Health Data and Knowledge Management 2008*. Ed. by P Yu P et al. Australia: Australian Computer Society, 2008, pp. 27–33. URL: https://eprints.qut.edu.au/30563/.

[13] Sung-Hwa Han et al. "Container Image Access Control Architecture to Protect Applications". In: *IEEE Access* 8 (2020), pp. 162012–162021. DOI: 10.1109/ACCESS.2020.3021044.

[14] Yuqiong Sun et al. "Security Namespace: Making Linux Security Frameworks Available to Containers". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1423–1439. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/sun.

[15] Delwar Alam et al. "Study of the Dirty Copy on Write, a Linux Kernel memory allocation vulnerability". In: *2017 International Conference on Consumer Electronics and Devices (ICCED)*. 2017, pp. 40–45. DOI: 10.1109/ICCED.2017.8019988.

[16] Hong Lan and Xuan Wang. "Research and Design of Concurrent Web Server on Linux System". In: *2012 International Conference on Computer Science and Service System*. 2012, pp. 734–737. DOI: 10.1109/CSSS.2012.188.

[17] Marcelo Amaral et al. "Performance Evaluation of Microservices Architectures Using Containers". In: *2015 IEEE 14th International Symposium on Network Computing and Applications*. 2015, pp. 27–34. DOI: 10.1109/NCA.2015.49.

[18] Zhanibek Kozhirbayev and Richard O. Sinnott. "A performance comparison of container-based technologies for the Cloud". In: *Future Generation Computer Systems* 68 (2017), pp. 175–182. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2016.08.025. URL: https://www.sciencedirect.com/science/article/pii/S0167739X16303041.

[19] Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[20] Ethan G. Young et al. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotcloud19/presentation/young.

[21] Eric Steven Raymond. *The Cathedral and the Bazaar*. O'Reilly Media, Inc., 2002. ISBN: 9780596001087.

[22] Xing Gao et al. "ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds". In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2017, pp. 237–248. DOI: 10.1109/DSN.2017.49.